Japanese Patent Laid-open Publication No. H9-288564

Publication date: November 4, 1997

Applicant: SAKAMURA TAKESHI, MITSUBISHI ELECTRIC CORP

Title: DATA PROCESSING DEVICE

【Partial Translation: from Page 98, Right column, Line 16 to Line 34 】

[Mnemonic]

TRAP

[Function of instruction]

TRAP conditionally

Conditioned software interruption, TRAP

[Instruction option]

/Various condition specifications (cccc)

[Instruction bit pattern and assembler expression]

Shown in Fig. 270

[Flag change]

Shown in Fig. 271

[Explanation] When a specified condition is met, an internal interruption (trap) is performed. Since EIT is activated by the TRAP instruction, ring 0 is always entered. The condition is specified in the same manner as that for the Bcc instruction. According to TRAP and TRAPA, a part of PSS and only PRNG of PSM are updated, similarly to other EIT processings. Fields (including PSB) other than PRNG of PSM are not updated. When a condition undefined by TRAP is specified, it will be a reserved instruction exception (RIE).

Fig. 270

TRAP

cccc is condition specification

Fig. 271

Instruction	F	X	V	L	M	Z	Comment
TRAP	—	—	—	—	—	—	

PATENT ABSTRACTS OF JAPAN

(11)Publication number : 09-288564
(43)Date of publication of application : 04.11.1997

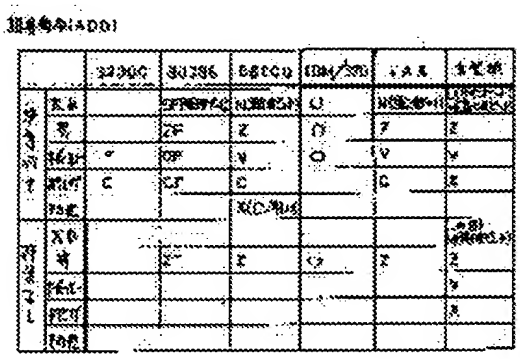
(51)Int.Cl. G06F 7/38
G06F 7/02
G06F 7/50
G06F 9/305

(21)Application number : 08-241822
(22)Date of filing : 17.06.1996

(71)Applicant : SAKAMURA TAKESHI
MITSUBISHI ELECTRIC CORP
(72)Inventor : SAKAMURA TAKESHI

(54) DATA PROCESSOR

(57)Abstract:
PROBLEM TO BE SOLVED: To easily and mathematically interpret an operation result by showing the occurrence of the overflow of the operation result stored in a destination operand even if an arithmetic operation which the instruction codes of a binary number with code and the binary number without code designate is executed.
SOLUTION: A program expressed by an instruction set where the first instruction code designating an instruction for executing the arithmetic operation of addition of subtraction between two operands as the binary number with code by means of the expression of two complements and a second instruction code designating an instruction for execution the same operation as the arithmetic operation between the two operands as the binary number without code, which is absolute value-expressed, are prepared is decoded. Then, the arithmetic operations which the instruction codes designate are executed. Information showing the occurrence of overflow in the operation result stored in the destination operand by means of the respective operations is expressed by the V flag.



LEGAL STATUS

[Date of request for examination] 17.06.1996
[Date of sending the examiner's decision of rejection] 23.05.2000
[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]
[Date of final disposal for application]
[Patent number]
[Date of registration]
[Number of appeal against examiner's decision of rejection]
[Date of requesting appeal against examiner's decision of rejection]
[Date of extinction of right]

(19)日本国特許庁（ＪＰ）　(12)公開特許公報（A）　(11)特許出願公開番号

特開平9－288564

(43)公開日　平成９年(1997)11月４日

(51)Int.Cl.[6]	識別記号	庁内整理番号	ＦＩ		技術表示箇所
Ｇ０６Ｆ	7/38		Ｇ０６Ｆ	7/38	Ｙ
	7/02			7/02	Ｊ
	7/50			7/50	Ｑ
	9/305			9/30	３４０Ａ

審査請求　有　発明の数３　書面　（全252頁）

(21)出願番号　特願平8－241822

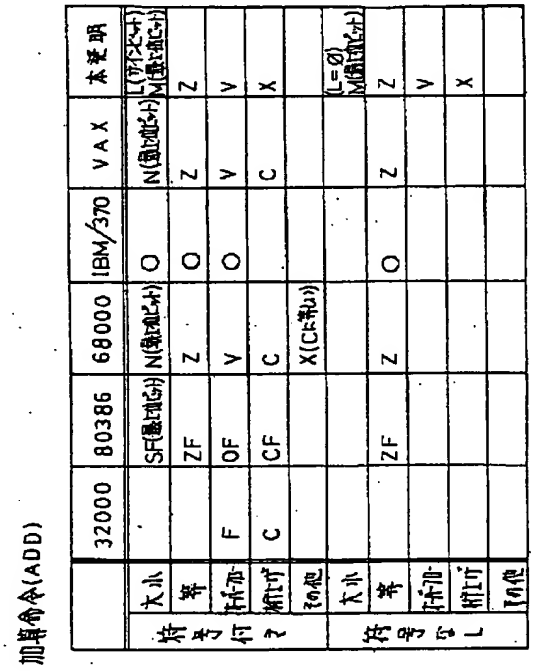
(22)出願日　平成８年(1996)６月17日

特許法第30条第１項適用申請有り　昭和62年３月31日頒布の「ＴＲＯＮシンポジウム資料集」に発表

(71)出願人　593084797
坂村　健
東京都港区白金台３－12－30－105
(71)出願人　000006013
三菱電機株式会社
東京都千代田区丸の内二丁目２番３号
(72)発明者　坂村　健
東京都港区白金台３－12－30－105
(74)代理人　弁理士　宮田　金雄　（外３名）

(54)【発明の名称】　データ処理装置

(57)【要約】
【課題】　符号付き算術演算と符号なし算術演算とを実行してもその算術結果の数学的解釈を容易に行えるようにする。
【解決手段】　２つのオペランドの間で２の補数表現による符号付き二進数として加算する演算を指定する加算命令ＡＤＤ、及び２つのオペランドの間で絶対値表現された符号なし二進数として加算する演算を指定する加算命令ＡＤＤＵが命令セットとして用意され、それぞれの加算演算結果のオーバーフローが発生したことを意味するフラッグをＶフラッグで表現したものである。



1

【特許請求の範囲】

【請求項１】　２つのオペランド間の加算又は減算の算術演算を２の補数表現による符号付き二進数として行う命令を指定する第１の命令コードと、２つのオペランドの間の前記算術演算と同一の演算を絶対値表現された符号なし二進数として行う命令を指定する第２の命令コードとが用意された命令セットにより表現されるプログラムを解読して、前記第１および第２の命令コードの各々が指定する算術演算を実行するデータ処理装置であって、

前記第１および第２の命令コードの各々で指定される命令によりデスティネーションオペランドに格納された演算結果におけるオーバーフローの発生を示す情報をフラッグで表現すべくなしてあることを特徴とするデータ処理装置。

【請求項２】　２つのオペランド間の加算又は減算の算術演算を２の補数表現による符号付き二進数として行う命令を指定する第１の命令コードと、２つのオペランドの間の前記算術演算と同一の演算を絶対値表現された符号なし二進数として行う命令を指定する第２の命令コードとが用意された命令セットにより表現されるプログラムを解読して、前記第１および第２の命令コードの各々が指定する算術演算を実行するデータ処理装置であって、

前記第１および第２の命令コードの各々で指定される命令によりなされた算術演算の演算結果の正負を示す情報をフラッグで表現すべくなしてあることを特徴とするデータ処理装置。

【請求項３】　２つのオペランドの間で２の補数表現による符号付き二進数としてその大小を比較する比較演算を指定する第１の命令コードと、２つのオペランドの間で絶対値表現された符号なし二進数としてその大小を比較する比較演算を指定する第２の命令コードとが用意された命令セットにより表現されるプログラムを解読し、前記第１および第２の命令コードの各々が指定する比較演算を実行するデータ処理装置であって、

前記第１および第２の命令コードの各々に従って実行された比較演算によるそれぞれの数学的大小関係の情報をフラッグで表現すべくなしてあることを特徴とするデータ処理装置。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】本発明はデータ処理装置に関し、演算結果の数学的解釈を容易にしたデータ処理装置を提案するものである。

【０００２】

【従来の技術及び発明が解決しようとする課題】従来のデータ処理装置における算術演算とフラッグ変化の関係についてみると符号付き二進数と符号なし二進数とを１種類の命令で演算し、オペランドを符号付き数と考えて

2

変化するフラッグと、符号なし数と考えて変化するフラッグとを各別に持っている。ところがこのような装置では、オペランドを符号つき数と考えた場合及び符号なし数と考えた場合の両方の演算結果の状態をフラッグに反映するため、オペランドを符号つき数と考えた場合に独立に意味をもつフラッグ、或はオペランドを符号なし数と考えた場合に独立に意味をもつフラッグが少なくなる。例えば、符号つき加算命令でオーバーフローが起こった場合でも結果の正負を正しく示すフラッグはなかった。また符号なし減算命令でオーバーフローが起こった場合も結果の正負を正しく示すフラッグはなかった。また独立に意味をもつフラッグが少ないため、条件によっては２つ以上のフラッグを同時に見ないと状態を判断できないという難点がある。

【０００３】次に従来装置においてデスティネーションが小さいときのフラッグ変化についてみると整数のビット列の演算結果又はソースオペランドサイズよりデスティネーションオペランドのサイズが小さいとき格納されなかったビットは無視する。又浮動小数点データの演算結果又はソースオペランドサイズよりデスティネーションオペランドのサイズが小さいとき格納されなかった下位ビットは一定規則により丸められる。このような装置には、整数演算の場合、小さなデスティネーションに格納しても値が正しく保てたかどうかをチェックするため、格納前に余分な演算を必要とした。またビット列演算の場合、格納するビット又は格納しないビットがバイト境界以外になる場合もありチェックが特に複雑となる。本発明は斯かる問題点を解決するためになされたもので、特に算術演算とフラッグ変化を密に対応づけ、演算結果の数学的解釈を容易に行えるようにしたデータ処理装置を提供することを目的とする。

【０００４】

【課題を解決するための手段】本発明は、以上の問題点を解決するために次のような手段をとっている。

１．符号つき演算と符号なし演算とを別命令にする。

２．全フラッグの変化を符号つき命令、符号なし命令に従い符号つき二進数として意味のあるもの、符号なし二進数として意味のあるものに専念させる。

３．演算結果、転送結果がデスティネーションサイズに格納できないとき数学的に正しい値が保たれているかに従い、フラッグを変化させる。

【０００５】本発明に係る第１のデータ処理装置は、２つのオペランド間の加算もしくは減算の算術演算を２の補数表現による符号付き二進数として行う命令を指定する第１の命令コードと、２つのオペランド間の加算もしくは減算の算術演算を絶対値表現された符号なし二進数として行う命令を指定する第２の命令コードとのいずれが指定する算術演算を行った場合も、デスティネーションオペランドに格納された演算結果のオーバーフローの発生をフラッグで示すことを特徴とする。

3

【０００６】本発明に係る第２のデータ処理装置は、２つのオペランド間の加算もしくは減算の算術演算を２の補数表現による符号付き二進数として行う命令を指定する第１の命令コードと、２つのオペランド間の加算もしくは減算の算術演算を絶対値表現された符号なし二進数として行う命令を指定する第２の命令コードとのいずれが指定する算術演算を行った場合も、算術演算の演算結果の正負をフラッグで示すことを特徴とする。

【０００７】本発明に係る第３のデータ処理装置は、２つのオペランド間で２の補数表現による符号付き二進数としてその大小を比較する比較演算を指定する第１の命令コードと、２つのオペランド間で絶対値表現された符号なし二進数としてその大小を比較する比較演算を指定する第２の命令コードとのいずれが指定する比較演算を行った場合も、２つのオペランドの数学的大小関係をフラッグで示すことを特徴とする。

【０００８】

【発明の実施の形態】以下、本発明の実施の形態につき詳しく説明する。以下の実施の形態では、

１．四則演算、比較命令、固定長ビット操作命令をすべて符号つきと符号なしの両方用意した。

２．Ｌフラッグにより、符号つき加算命令と符号つき減算命令の結果の正負をオーバーフローに関係なく示す。

３．Ｖフラッグにより結果が格納できなかったときの数学的意味（符号つき、符号なしに応じて）を示す。

従来のプロセッサ即ち３２０００，８０３８６，６８０００，ＩＢＭ／３７０，ＶＡＸでは、図４０９、図４１０、図４１１、図４１２、図４１３、図４１４に示すように算術演算とフラッグ変化の関係を見ると、ほとんど全てのプロセッサが符号付き数と符号なし数を１種類の命令で演算し、オペランドを符号付き数と考えて変化するフラッグと、符号なし数と考えて変化するフラッグとを共通に持っていた。

【０００９】従来のプロセッサで符号なし加算命令が存在しない場合は、符号付き加算命令として演算を実行した結果のフラッグ情報を調べることによって、得られた結果が符号なし数として有効であるか否かを判断している。図４１５に示すように、８ビット符号なし数として１９２（１０）、８ビット符号付き数として－６４（１０）を各々加算する場合に符号なし数として各々を加算した結果はオーバーフローを生じるはずであるが、符号付き数として加算された場合の結果は最上位からの桁上げビットと最上位ビットへの桁上げが一致するためオーバーフローを生じない。従って、符号なし数としてオーバーフローを調べるには、他のフラッグ、ＶＡＸの場合は桁上げフラッグがセットされていることにより検出される。但し、この場合でもオーバーフローフラッグがリセットされていることと桁上げフラッグがセットされていることの２個の条件を調べなければならず、プログラム数が増え効率が悪くなる。一方、本発明では符号付き

4

と符号なしの両方の命令を備えており、各々の実行結果でフラッグに反映されるので、符号なし数として扱いたい場合は符号なし加算命令を実行し、その結果オーバーフローを示すフラッグのみを参照すればオーバーフローを生じているかどうかを検出できる。また従来のプロセッサでは符号付き加算命令を実行した場合、オーバーフローが生じない時は結果の最上位ビットを見ることにより演算結果の正負が判断できたが、オーバーフローが生じた場合は結果の最上位ビットを見ただけでは演算結果の正負が判断できず、つまり結果としてのデータが無効であるという理由からオーバーフローを優先させ演算結果を無視していた。例えば図４１６に示すように、８ビットの負の最大数同士（－１２８（１０））を加算した結果オーバーフローが検出されるが、結果の最上位ビットは０であり（最上位ビットからの桁上げを除く）結果が負であるということに反している。本発明では、オーバーフローが生じた時にでも演算結果の正負を情報として提供するフラッグが備わっており、そのビットを参照するだけで処理結果が明瞭である。

【００１０】更に、本発明では各演算におけるオペランドサイズが異種サイズ間でも演算実行され、異種サイズ間どうしの演算においてもオーバーフロー検出される。例えば、演算結果がデスティネーションオペランドのデータサイズを越えてしまうような場合、又、ビット抽出命令によって取り出されたデータがデスティネーションオペランドのデータサイズを越えてしまうような場合にもオーバーフローを示すフラッグがセットされる。また、図４１７に示すように異種サイズ間どうしの符号付き加算演算においてオーバーフローが生じた場合に、デスティネーションオペランドに結果として保持されたデータ（ｃ）の最上位ビット（（ｄ）で示される）以外に、（ａ）で示される演算結果の最上位ビット、つまり（ｂ）で示されるサインビットが、正負を示すフラッグに情報として格納される（この場合は正を表す。）ので、たとえオーバーフローが生じて数値データとして意味を持たなくても演算結果のデータの正負の真偽が検出できる。

【００１１】次に本発明における符号付き演算の場合の例について説明する。

ADD RO.H, R1.B

この命令は、オペランドＲ０をオペランドＲ１に符号付きで足し込む。．Ｈは、オペランドＲ０がハーフワード（１６ビット）であることを示す。．Ｂは、オペランドＲ１がバイト（８ビット）であることを示す。

－－－－－－－－

この命令の実行例を次に示す。

5

R0:　0000.0000.0111.1111

（＝+127　16 ビット符号付き）

＋

R1:　0000.0001

（＝+1　8 ビット符号付き）

‖

演算結果:　00 ……0.1000.0000

（＝+128　2 進数符号付き）

↓

演算結果:　1000.0000

（＝-128　8 ビット符号付き）

即ち、８ビット符号付きでは演算結果を表現できない。
こういった場合にＶ－ｆｌａｇは１になる。

－－－－－－－－－

この命令の他の実行例を次に示す。

R0:　1111.1111.0111.1111

（＝-129　16 ビット符号付き）

＋

R1:　0000.0001

（＝+1　8ビット符号付き）

R0:　0000.0000.1111.1111

（＝+255　16 ビット符号なし）

＋

R1:　0000.0001

（＝+1　8 ビット符号なし）

‖

演算結果:　00 ……001.0000.0000

（＝+256　2 進数符号なし）

↓

演算結果:　0000.0000

（＝0　8 ビット符号なし）

即ち、８ビット符号なしでは演算結果を表現できない。
こういった場合にＶ－ｆｌａｇは１になる。

－－－－－－－－－

この命令の他の実行例を次に示す。

6

‖

演算結果:　11 ……1.1000.0000

（＝-128　2進数符号付き）

↓

演算結果:　1000.0000

（＝-128　8ビット符号付き）

即ち、８ビット符号付きで演算結果を表現できる。こういった場合にＶ－ｆｌａｇは０になる。これは後のＡＤＤ命令の項を参照。

【００１２】次に符号なし演算の場合の例について説明する。

ADDU R0.H, R1.B

この命令は、オペランドＲ０をオペランドＲ１に符号なしで足し込む。．Ｈは、オペランドＲ０がハーフワード（１６ビット）であることを示す。．Ｂは、オペランドＲ１がバイト（８ビット）であることを示す。

－－－－－－－－－

この命令の実行例を次に示す。

7

R0: 0000.0000.0111.1111
（＝+127 16 ビット符号なし）
＋
R1: 0000.0001
（＝+1 8 ビット符号なし）
‖
演算結果: 00 ……0.1000.0000
（＝+128 2 進数符号なし）
↓
演算結果: 1000.0000
（＝+128 8 ビット符号なし）

即ち、８ビット符号なしで演算結果を表現できる。こういった場合にＶ－ｆｌａｇは０になる。これは後のＡＤＤＵ命令の項を参照。次に符号付き転送の場合の例について説明する。

```
BFINS src.H, #2, #3, base.H
```

この命令は、オペランドｓｒｃの値を、オペランドｂａｓｅの中のビット番号２から４の始まる３ビット長のビットフィールドに符号付きで転送する。．Ｈは、オペランドｓｒｃ，ｂａｓｅがハーフワード（１６ビット）であることを示す。

－－－－－－－－－

この命令の実行例を次に示す。

【００１３】

【数１】

src: 1111,1111,1111,1111
（＝-1 16ビット符号付き）
↓
転送されるビットフィールド:
111
（＝-1 3ビット符号付き）
base: 00__,_000,0000,0000
（下線部は転送部位を示す）
↓
転送結果: 0011,1000,0000,0000

【００１４】即ち、３ビット符号付きでは演算結果を表現できない。こういった場合にＶ－ｆｌａｇは１になる。

－－－－－－－－－

この命令の他の実行例を次に示す。

【００１５】

【数２】

8

src: 1111,1111,1111,1111
（＝-1 16ビット符号付き）
↓
転送されるビットフィールド:
111
（＝-1 3ビット符号付き）
base: 00__,_000,0000,0000
（下線部は転送部位を示す）
↓
転送結果: 0011,1000,0000,0000

【００１６】即ち、この場合は３ビット符号付きで演算結果を表現できる。こういった場合にＶ－ｆｌａｇは０になる。これは後のＢＦＩＮＳ命令の項を参照。

【００１７】次に符号なし転送の場合の例について説明する。

```
BFINSU src.H, #2, #3, base.H
```

この命令は、オペランドｓｒｃの値を、オペランドｂａｓｅの中のビット番号２から４の始まる３ビット長のビットフィールドに符号なしで転送する。．Ｈは、オペランドｓｒｃ，ｂａｓｅがハーフワード（１６ビット）であることを示す。

－－－－－－－－－

この命令の実行例を次に示す。

【００１８】

【数３】

src: 0000,0000,0000,1111
（＝+15 16ビット符号なし）
↓
転送されるビットフィールド:
111
（＝+7 3ビット符号なし）
base: 00__,_000,0000,0000
（下線部は転送部位を示す）
↓
転送結果: 0011,1000,0000,0000

【００１９】即ち、３ビット符号なしでは演算結果を表現できない。こういった場合にＶ－ｆｌａｇは１になる。

－－－－－－－－－

この命令の他の実行例を次に示す。

【００２０】

【数４】

src:　0000, 0000, 0000, 0011

（＝+3　16ビット符号なし）

↓

転送されるビットフィールド:

011

（＝+3　3ビット符号なし）

base:　00__, _000, 0000, 0000

（下線部は転送部位を示す）

↓

転送結果: 0001, 1000, 0000, 0000

【００２１】即ち、この場合は３ビット符号付きで演算結果を表現できる。こういった場合にＶ－ｆｌａｇは０になる。これは後のＢＦＩＮＳＵ命令の項を参照。

【００２２】次に本発明におけるフラッグの構成を述べる。

Ｐ・・・Ｐビットエラーフラッグ

Ｆ・・・ストリング命令やキュー命令等の高機能命令の終了条件を示す

Ｘ・・・多倍長計算用の桁上がりを示す

Ｖ・・・オーバーフローが発生したことを示す

Ｌ・・・比較命令等において第１オペランドの方が小さいことを示す

Ｍ・・・演算結果のＭＳＢ（Ｍｏｓｔ　Ｓｉｇｎｉｆｉｃａｎｔ　Ｂｉｔ）が１であることを示す

Ｚ・・・演算結果が０になったことを示す

従来のプロセッサで使用するＣａｒｒｙフラッグは、符号なし整数の大小関係を表すという意味と、多倍長演算の桁上がりを表すという意味がある。しかし、本発明ではＸフラッグを備えているためＣａｒｒｙフラッグは整数の大小関係を表すという意味でのみ用いられる。従って、本発明ではこのＣａｒｒｙフラッグを大小関係を表すフラッグであると定義し、名前をＬフラッグとしている。Ｌフラッグは符号なし演算の場合には従来のＣａｒｒｙフラッグと同じ振る舞いをするが、符号付き演算の場合には従来のＣａｒｒｙフラッグとは異なり、オーバーフローまで考慮した真の大小関係を意味する。Ｘフラッグは、多倍長の演算を行う場合に桁下がりの状態を保持するために使用される。符号付き演算でも符号なし演算の時と同様な変化をする。これは、従来のプロセッサのＣａｒｒｙフラッグとほぼ同じ意味をなす。従来のプロセッサではシフト命令等ではみ出したビットを入れるためにＣａｒｒｙフラッグを用いていたが、本発明ではＬフラッグを設けているため、はみ出したビットはＸフラッグに入れることにしている。Ｖフラッグは、演算の結果がデスティネーションオペランドで指定されたサイズでは表現できなかったことを示す。つまり、デスティネーションオペランドのサイズの符号付き整数で表現できない時にもＶフラッグがセットされる。Ｍフラッグと

Ｚフラッグは演算結果をデスティネーションオペランドのサイズに変換した後の値を基準にして変化する。従って、ソースオペランドのサイズよりも、デスティネーションオペランドのサイズの方が小さい場合は演算結果が０でなくてもＺフラッグがセットされることがありうる。以下、符号付きと符号なしの両方を持つ演算命令各々についてのフラッグの変化について述べる。

【００２３】（１）ＡＤＤ命令（符号付き）とＡＤＤＵ命令（符号なし）

Ｌフラッグ：演算結果が負になったことを示す。ＡＤＤＵ命令では０にリセットされる。

Ｍフラッグ：演算の結果、デスティネーションオペランドに格納された最上位ビットを示す。Ｍフラッグも正負を表すが、オーバーフローを生じた場合正しい正負を表示しているとは言えない。

Ｚフラッグ：演算の結果、デスティネーションオペランドに格納された値が０であることを示す。

Ｖフラッグ：デスティネーションサイズを越えるような演算結果になったことを示す。

Ｘフラッグ：デスティネーションサイズを越えて桁上げが生じたことを示す。

（２）ＳＵＢ命令（符号付き）とＳＵＢＵ命令（符号なし）

Ｌフラッグ：演算結果が負になったことを示す。

Ｍフラッグ：演算の結果、デスティネーションオペランドに格納された最上位ビットを示す。Ｍフラッグも正負を表すが、オーバーフローを生じた場合正しい正負を表示しているとは言えない。

Ｚフラッグ：演算の結果、デスティネーションオペランドに格納された値が０であることを示す。

Ｖフラッグ：デスティネーションサイズを越えるような演算結果になったことを示す。ＳＵＢＵ命令では、結果が負になった場合も相当する。

Ｘフラッグ：デスティネーションサイズを越えて桁下げが生じたことを示す。

【００２４】（３）ＭＵＬ命令（符号付き）とＭＵＬＵ命令（符号なし）

Ｌフラッグ：演算結果が負になったことを示す。ＭＵＬＵ命令では０にリセットされる。

Ｍフラッグ：演算の結果、デスティネーションオペランドに格納された最上位ビットを示す。Ｍフラッグも正負を表すが、オーバーフローを生じた場合正しい正負を表示しているとは言えない。

Ｚフラッグ：演算の結果、デスティネーションオペランドに格納された値が０であることを示す。

Ｖフラッグ：デスティネーションサイズを越えるような演算結果になったことを示す。

Ｘフラッグ：不変

（４）ＤＩＶ命令（符号付き）とＤＩＶＵ命令（符号なし）

11

Ｌフラッグ：演算結果が負になったことを示す。ＤＩＶＵ命令では０にリセットされる。ただし、ＤＩＶ命令において（最大負数）÷（－１）は実行した場合は０にリセットされる。又、零除算例外発生時のＬフラッグは不変である。

Ｍフラッグ：演算の結果、デスティネーションオペランドに格納された最上位ビットを示す。Ｍフラッグも正負を表すが、オーバーフローを生じた場合、つまりＤＩＶ命令における（最大負数）÷（－１）を実行した際は１にセットされ、零除算例外発生時は不変である。

Ｚフラッグ：演算の結果、デスティネーションオペランドに格納された値が０であることを示す。ただし、ＤＩＶ命令における（最大負数）÷（－１）は実行した際は０にリセットされ、零除算例外時は不変である。

Ｖフラッグ：デスティネーションサイズを越えるような演算結果、つまりＤＩＶ命令における（最大負数）÷（－１）の実行、及び、零除算例外を発生した場合にのみ１にセットされる。

Ｘフラッグ：不変

【００２５】（５）ＭＯＶ命令（符号付き）とＭＯＶＵ命令（符号なし）

Ｌフラッグ：不変

Ｍフラッグ：演算の結果、デスティネーションオペランドに格納された最上位ビットを示す。Ｍフラッグも正負を表すが、オーバーフローを生じた場合正しい正負を表示しているとは言えない。

Ｚフラッグ：演算の結果、デスティネーションオペランドに格納された値が０であることを示す。

Ｖフラッグ：デスティネーションサイズを越えるような演算結果になったことを示す。（符号ビット拡張部のはみ出しは許される）

Ｘフラッグ：不変

（６）ＲＥＭ命令（符号付き）とＲＥＭＵ命令（符号なし）

Ｌフラッグ：演算結果が負になったことを示す。（演算前のデスティネーションオペランドの符号をそのまま維持する）ＲＥＭＵ命令では０にリセットされる。ただし、零除算例外発生時は不変である。

Ｍフラッグ：演算の結果、デスティネーションオペランドに格納された最上位ビットを示す。正負を示すフラッグであるが、零除算例外発生時は不変である。

Ｚフラッグ：演算の結果、０になったことを示す。ただし、零除算例外発生時は不変である。

Ｖフラッグ：０にリセットされる。零除算例外が生じても剰余がオーバーフローするわけではないのでクリアしておくことにすると、例外処理中でＤＩＶ命令によるエラーか、ＲＥＭ命令によるエラーかが判別できる。

Ｘフラッグ：不変

【００２６】（７）ＣＭＰ命令（符号付き）とＣＭＰＵ命令（符号なし）

12

Ｌフラッグ：演算の結果、オペランド１がオペランド２よりも小さいことを示す。

Ｍフラッグ：不変

Ｚフラッグ：演算の結果、オペランド１とオペランド２が等しいことを示す。（オペランドサイズが異なる場合、サイズの大きい方に符号拡張又は、ゼロ拡張されて実行される。）

Ｖフラッグ：不変

Ｘフラッグ：不変

（８）ＢＦＣＭＰ命令（符号付き）とＢＦＣＭＰＵ命令（符号なし）

Ｌフラッグ：演算の結果、ビットフィールド値が小さいことを示す。

Ｍフラッグ：不変

Ｚフラッグ：演算の結果、ソース値とビットフィールドが等しいことを示す。

Ｖフラッグ：不変

Ｘフラッグ：不変

【００２７】（９）ＢＦＥＸＴ命令（符号付き）とＢＦＥＸＴＵ命令（符号なし）

Ｌフラッグ：不変

Ｍフラッグ：演算の結果、デスティネーションオペランドに格納された最上位ビットを示す。正負を示すフラッグであるが、オーバーフローを生じた場合正しい正負を表示しているとは言えない。

Ｚフラッグ：演算の結果、デスティネーションオペランドに格納された値が０であることを示す。

Ｖフラッグ：デスティネーションサイズを越えるような演算結果になったことを示す。ただし、抽出されるデータが、ＢＦＥＸＴ命令の場合に符号拡張されたデータであり拡張された部分のみがサイズをオーバーしている時はＶフラッグはセットされない。同様に、ＢＦＥＸＴＵ命令の場合に抽出データがゼロ拡張されたデータであり拡張された部分のみがサイズをオーバーしている時もＶフラッグはセットされない。

Ｘフラッグ：不変

（１０）ＢＦＩＮＳ命令（符号付き）とＢＦＩＮＳＵ命令（符号なし）

Ｌフラッグ：不変

Ｍフラッグ：演算の結果、データ挿入フィールドに格納された最上位ビットを示す。正負を示すフラッグであるが、オーバーフローを生じた場合正しい正負を表示しているとは言えない。

Ｚフラッグ：演算の結果、データ挿入フィールドに格納された値が０であることを示す。

Ｖフラッグ：データ挿入フィールドのサイズを越えるような演算結果になったことを示す。ただし、ソースオペランドが、ＢＦＩＮＳ命令の場合に符号拡張されたデータであり拡張部分のみがサイズをオーバーしている時はＶフラッグはセットされない。同様に、ＢＦＩＮＳＵ命

13

令の場合にソースオペランドがゼロ拡張されたデータであり拡張された部分のみがサイズをオーバーしている時もＶフラッグはセットされない。

Ｘフラッグ：不変

以下本発明装置の全貌を詳細に説明する。説明が長大になるので目次を付し、また付加的説明、説明の整理のために付録を付けた。

【００２８】目　次

14

付録７．制御レジスタと制御空間
付録８．本発明装置のＣＴＸＢ
付録９．本発明装置のＥＩＴ処理
付録１０．本発明装置の命令セットパターン
付録１１．高機能命令の詳細仕様と終了時のレジスタ値
付録１２．オペランドが干渉した場合の動作
付録１３．キャッシュやＴＬＢの整合性確保について
【００２９】１．本発明装置の特徴
【００３０】１－１．基本設計思想
・本発明装置はＲＩＳＣではない。基本命令の高速実行を第一目標とし、さらに高機能命令を追加した。
・３２ビット版の本発明装置３２と６４ビット版の本発明装置６４を同時に設計し、３２ビット版と６４ビット版のチップをシリーズ化した。したがって、６４ビットデータ、６４ビットアドレッシングへの拡張が始めから考慮されている。
・ＯＳと込みで開発し、リアルタイムＯＳであるＩＴＲＯＮ（Ｉｎｄｕｓｔｏｒｉａｌ－ＴＲＯＮ）とワークステーション用のＯＳであるＢＴＲＯＮ（Ｂｕｓｉｎｅｓｓ－ＴＲＯＮ）を高速で実行することを目指した。本発明装置はＴＲＯＮ〈〈ＬＩＲ〉〉仕様を満たし、特に実記憶環境での高速処理に重点を置く。
・将来のＡＳＩＣ　ＬＳＩの核となるマイクロプロセッサとする。
【００３１】１－２．ＯＳ向きアーキテクチャ
・ビットマップ操作サポート命令
ＢＴＲＯＮで必要となるビットマップの移動、演算を行なう命令
・コンテキストスイッチ命令
ＩＴＲＯＮにおいて高速のタスク切り替えを行なうための命令
・キュー操作命令
ＩＴＲＯＮのレディキュー、ウエイトキューの操作を行なう命令
・２レベルのリング保護によるメモリ管理（将来の拡張用にさらに２レベルのリングを用意している。）
【００３２】１－３．チューニングされた命令セット
・頻度の高い命令、アドレッシングモードが短い命令となるようにチューニングレジスタ間演算、リテラル演算の命令長を短縮
【００３３】１－４．コンパイラ向きの命令セット
・直交化された命令セット
・データの保持、アドレスの保持、インデクス値の保持といった各種の目的に使用できる１６本の汎用レジスタ
・強力なアドレッシングモード
付加モードにより、任意段数のインデクス加算と間接参照が可能。
・異なるデータサイズ間での演算が可能
ソースオペランドとデスティネーションオペランドのサイズを別々に指定可能。

・高級言語向きの高機能ジャンプ命令
【００３４】２．本発明装置３２と本発明装置６４
本発明装置では、３２ビット版の本発明装置３２と６４ビット版の本発明装置６４をシリーズ化して扱っており、６４ビット版への拡張が始めから考慮されているのが大きな特徴である。本発明装置６４では、６４ビットのリニアアドレスの空間が提供される。また、本発明装置６４では、本発明装置３２で扱うデータタイプに加えて、６４ビット整数を扱うことができる。本発明装置６４での３２ビット／６４ビット切り替え方法は次のようになっている。
・オペランドのデータサイズについて
各命令、各オペランド毎に存在するサイズ指定ビットにより、命令単位、オペランド単位で３２ビット／６４ビットを選択する。データサイズの場合は、３２ビット、６４ビットのほかに８ビット、１６ビットも利用できるので、４つのサイズの選択を２ビットのフィールドで指定する。本発明装置３２では、６４ビットデータを扱わず、６４ビットのデータサイズを指定した命令はエラーとする。
・ポインタのサイズについて
通常は本発明装置３２で３２ビットポインタ、本発明装置６４で６４ビットポインタを使用するが、本発明装置６４で本発明装置３２用のオブジェクトコードを実行するため、本発明装置６４にはポインタサイズを３２ビットにするモードを設ける。このモードはＰＳＷ中で指定されるので、コンテキスト（プロセスやタスク）単位で３２ビット用のプログラムと６４ビット用のプログラムを混在させることは可能である。このほか、６４ビットアドレッシングを行なうための拡張用ビットとして、メモリアクセスを伴なうオペランド毎に「Ｐビット」と呼ばれる予約ビットが設けられている。ポインタサイズの３２ビット／６４ビット切り替えを命令単位とせず、モードにしたのは、次のような理由による。ポインタの場合、異種のサイズのものを混在させることは本質的に無理がある。というのは、ポインタというのは場所を区別するものであるため、一つでも６４ビットのものがあれば、全体を６４ビットのポインタにしないと区別できないからである。したがって、３２ビットポインタと６４ビットポインタを命令毎に切り替え、混在できるようにしたとしても、大部分はコンテキスト単位で同じ指定を繰り返すだけになり、ビット割り当てとしては効率の悪いものになる。そのような状況では、モードの方が適当である。
【００３５】モードを使って３２／６４を切り替える場合、モードをいつ設定するか、本発明装置３２と本発明装置６４の互換性が大丈夫かといった疑問が生ずるかもしれない。しかし、デフォルトが３２ビットポインタとなるようにしておき、６４ビットアドレスを使用する時にモード変更するという形態にすれば、本発明装置６４

でも本発明装置３２用のプログラムをそのまま走らせることができる。また、３２ビットポインタと６４ビットポインタの切り替えをモードではなく命令単位にしたとしても、ＯＳがスタックをどこに設定するか、システムコールのパラメータが３２ビット／６４ビットのどちらか、といったことを判断するために、ＯＳは各コンテキストが３２ビットか６４ビットかということを認識しておく必要がある。その場合には、（スタックに退避された）ＰＳＷ中のモードを見て３２ビット／６４ビットを判断できる方が便利なことがある。

【００３６】３．本発明装置仕様のクラス分け

本発明装置では、６４ビット版への拡張、シリーズ化、多様な用途への適応、などといった要求に対応するため、インプリメントするかどうかオプションとなっている機能がある。このような「オプション機能」の位置付けを明確にするため、本発明装置の仕様を次のようにクラス分けする。

〈〈Ｌ０〉〉仕様（Ｌｅｖｅｌ　０）

本発明装置として必ず満たさなければならない仕様である。〈〈Ｌ０〉〉仕様の例は、ユーザプログラムから見たプログラミングモデル（ＩＳＰの大部分、汎用レジスタ、ＰＳＨ）、機械語のビットパターンなどである。仕様書中では、特に何も書いてない部分が〈〈Ｌ０〉〉仕様となる。

〈〈Ｌ１〉〉仕様（Ｌｅｖｅｌ　１）

インプリメントしておくのが原則であるが、特に用途の限定された軽い仕様のプロセッサを作りたい場合には、必ずしもインプリメントしなくてもよい仕様である。

〈〈Ｌ１〉〉仕様になるのは、ストリング命令、付加モード、キュー操作命令、ビットマップ命令といった高機能命令などであるが具体的にどの命令を〈〈Ｌ１〉〉仕様とするかは別に定める。

【００３７】〈〈Ｌ１Ｒ〉〉仕様（Ｌｅｖｅｌ　１　Ｒｅａｌ）

〈〈Ｌ１〉〉仕様から命令再実行関係の機能とＭＭＵ関係の機能を削除した仕様であり、ＩＴＲＯＮとμＢＴＲＯＮなどを実記憶ベースで効率よく動かすための仕様である。〈〈Ｌ１Ｒ〉〉の命令セットは〈〈Ｌ１〉〉とほぼ同じであり、コンパイラやユーザプログラムは共通に使用できるようにする。ただし、ＭＭＵ関係（ＭＯＶＰＡなど）、ＯＳ関係（ＪＲＮＧなど）の一部の命令については、サポートしないものがある。

〈〈Ｌ２〉〉仕様（Ｌｅｖｅｌ　２）

将来のハードウエア量の増加にしたがって導入される予定の仕様である。命令の対称性を高めるための仕様と、演算の高速化に対応して新たに追加する命令の仕様とがある。前者の例としてはＢＶＳＣＨ命令の’／Ｂ’オプション、ストリング命令での複雑な終了条件、無限段数の付加モードなどがあり、後者の例としてはＩＮＤＥＸ命令などがある。仕様書中では、〈〈Ｌ２〉〉仕様の部分を〈〈Ｌ２〉〉で示す。

【００３８】〈〈ＬＸ〉〉仕様（ｅＸｔｅｎｓｉｏｎ）

本発明装置６４への拡張にしたがって導入される予定の仕様である。Ｌ２仕様と同様の意味を持つが、本発明装置６４への対応ということで別のクラスとして扱う。

〈〈ＬＸ〉〉仕様の例は、６４ビット演算命令などである。仕様書中では、〈〈ＬＸ〉〉仕様の部分を〈〈ＬＸ〉〉で示す。

〈〈ＬＵ〉〉仕様（Ｕｎｄｅｆｉｎｅｄ）

将来の拡張によって導入される予定の仕様であるが、現段階ではまだ具体的な仕様まで提示されていないものである。

〈〈ＬＶ〉〉仕様（Ｖａｒｉａｂｌｅ）

各メーカーが全く自由に仕様を決めてよい部分である。チップのピン配置、パイプラインの段階や性能に関する仕様、各メーカーに割り当てられた命令のビットパターン、制御レジスタの使い方などが〈〈ＬＶ〉〉仕様の例である。このうち、各メーカーに割り当てられた命令ビットパターンについては、ビットパターンのレファレンス中でＬＶｒｅｓｅｒｖｅｄにより示されている。

【００３９】〈〈ＬＡ〉〉仕様（Ａｌｔｅｒｎａｔｉｖｅ）

本発明装置としての標準仕様が提示されている（あるいは、提示される予定がある）が、他にやむを得ない理由があれば変更してもよいという仕様である。もちろん、仕様を変更した部分に関しては互換性の失われる場合がある。〈〈ＬＡ〉〉仕様は、ＴＲＯＮとしての互換性を保証しない仕様である。〈〈ＬＡ〉〉仕様の例は、メモリ管理方式、制御レジスタと特権命令の一部などであり、主にＯＳの関係する部分である。本発明装置はＭＭＵを内蔵せず、特に実記憶環境での高速処理を目指す。従ってメモリ管理に関する〈〈ＬＡ〉〉仕様の大部分は本発明装置でサポートされない。

【００４０】４．レジスタセット

・本発明装置３２では３２ビット長の汎用レジスタが１６本、本発明装置６４では６４ビット長の汎用レジスタが１６本存在する。

・スタックポインタ（Ｓｔａｃｋ　Ｐｏｉｎｔｅｒ－ＳＰ）、フレームポインタ（Ｆｒａｍｅ　Ｐｏｉｎｔｅｒ－ＦＰ）は汎用レジスタに含まれる。ＳＰはＲ１５、ＦＰはＲ１４となる。

・プログラムカウンタ（Ｐｒｏｇｒａｍ　Ｃｏｕｎｔｅｒ－ＰＣ）は汎用レジスタに含まれない。

・汎用レジスタは、データ保持、ベースアドレス保持、あるいはインデクスレジスタとして、各種の目的に使用できる。

・プロセッサの状態を保持するレジスタ（Ｐｒｏｃｅｓｓｏｒ　ＳｔａｔｕｓＷｏｒｄ－ＰＳＷ）を持つ。図１、図２は、本発明装置６４の場合〈〈ＬＸ〉〉のレジスタセットを示す。

19

・ＳＰはコンテキスト（リング番号、割り込み処理中）に応じて切り替わる。

・ＰＳＷは４バイトからなる。下位第一バイトがステータスの表示（Ｐｒｏｃｅｓｓｏｒ　Ｓｔａｔｕｓ　Ｂｙｔｅ－ＰＳＢ）、下位第二バイトがユーザのモード設定（ＰＳＢと合わせてＰｒｏｃｅｓｓｏｒ　Ｓｔａｔｕｓ　Ｈａｌｆｗｏｒｄ－ＰＳＨ）、上位の２バイトがシステムの状態表示用、となる。

・本発明装置はいわゆるｂｉｇ－ｅｎｄｉａｎのチップであり、レジスタ上のデータについては、８ビット、１６ビットのデータをＬＳＢ側に詰めて配置する。したがって、データサイズとは無関係な絶対的なビット番号を定義することができない。ビット番号を議論する場合には、必ずデータサイズと組にして扱う必要がある。これを「ビット位置」と呼ぶ。

・レジスタ上の８ビットデータに対しては、ビット位置はＭＳＢ側から０，１，．．．７と付けられる。また、レジスタ上の１６ビットデータに対しては、ビット位置はＭＳＢ側から０，１，．．．１５と付けられ、レジスタ上の３２ビットデータに対しては、ビット位置はＭＳＢ側から０，１，．．．３１と付けられる。したがって、８ビットデータのビット位置７のビット、１６ビットデータのビット位置１５のビット、３２ビットデータのビット位置３１のビットが物理的には同一のビットとなる。

・レジスタをデスティネーションオペランドとする命令において、レジスタ側のデータサイズが８ビット、１６ビットであった場合には、上位バイトは影響を受けず、無変化となる。これは、メモリ上で演算を行なった場合の仕様と合わせたものである。上位ビットにまで影響を与えたい場合は、異種サイズ間の演算を利用する。

［例］

```
MOV     #H'12345678, R0.W
MOV     #H'aa, R0.B
        R0=H'123456aaとなる。
```

・レジスタ上に８ビット、１６ビットのデータを置く場合には、ＬＳＢ側に詰められるので、例えば、

```
MOV.W   #H'12345678, R0
MOV.B   #H'aa, R0
MOV.W   R0, R1
```

の結果はＲ１＝Ｈ’１２３４５６ａａとなる。一方、メモリに対して同じことを行なった場合

```
MOV.W   #H'12345678, @R0
MOV.B   #H'aa, @R0
MOV.W   @R0, R1
```

には、８ビット、１６ビットのデータのＭＳＢ側が揃うことになるので、Ｒ１＝Ｈ’ａａ３４５６７８となる。

20

レジスタ上とメモリ上で結果が異なるので、注意が必要である。

【００４１】５．データタイプ

本発明装置では、いわゆるｂｉｇ－ｅｎｄｉａｎを採用している。すなわち、バイトアドレスの指定、ビット番号の指定とも、小さい番号（アドレス）の方がＭＳＢ（Ｍｏｓｔ　Ｓｉｇｎｉｆｉｃａｎｔ　Ｂｉｔ／Ｂｙｔｅ）となっている。ｂｉｇ－ｅｎｄｉａｎでは、メモリ上のあるデータについて、それを８ビットデータとして見る時と１６（３２）ビットデータとして見る時のアドレスが異なってくるため、注意が必要である。例えば、

address:	N	N+1	N+2	N+3
data:	0	0	0	H'12

といった場合に、３２ビットデータとしてのアドレスＮの内容はＨ’００００００１２であるが、（Ｈ’は１６進を表わす）、同じ内容のデータを８ビットデータとして扱うときは、アドレスＮ＋３を参照しなければならない。

【００４２】ただし、レジスタ上のデータに関しては、８ビットデータ、１６ビットデータがＬＳＢ側に詰めて配置されるため、レジスタ上に置かれたデータを、そのまま別のサイズのデータとして扱うことができる。例えば、

```
MOV     #0, R0.W
MOV     #H'12, R0.B
MOV     R0.W, R1.W
```

の結果はＲ１＝Ｈ’００００００１２となる。（命令の意味については本文参照）

一方、メモリに対して同じことを行なった場合

```
MOV     #0, @R0.W
MOV     #H'12, @R0.B
MOV     @R0.W, R1.W
```

には、８ビットデータＨ’１２と３２ビットデータのＭＳＢ側が揃うことになるので、Ｒ１＝Ｈ’１２０００００となる。本発明装置でサポートしているデータタイプを　以下に説明する。

【００４３】５－１．ビット

図３のように太線内が対象ビットである。メモリ上のビット操作の場合、ｏｆｆｓｅｔは任意レジスタ上のビット操作の場合、ｏｆｆｓｅｔは一つのレジスタ内に限定（ｏｆｆｓｅｔの上位ビットを無視する）

ビットの指定は、ｂａｓｅ＿ａｄｄｒｅｓｓの指定、ｂａｓｅ＿ａｄｄｒｅｓｓのサイズの指定、ｏｆｆｓｅｔの指定の組によって行なわれる。メモリ上のビットを対象とした場合には、ｂａｓｅ＿ａｄｄｒｅｓｓで示されるメモリアドレスのＭＳＢがｏｆｆｓｅｔ＝０のビットとなる。この時、ｂａｓｅ＿ａｄｄｒｅｓｓのサイズの

21

指定は、実際に操作されるビットには影響しない。ビット操作命令では、メモリに対してｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅを行なうアクセスサイズを指定するためにｂａｓｅ＿ａｄｄｒｅｓｓのサイズの指定が利用されるが、アクセスサイズが異なっても実際に操作されるビットは同じである。一方、レジスタ上のビットを対象とした場合には、ｂａｓｅ＿ａｄｄｒｅｓｓのサイズとして指定されたデータサイズでのＭＳＢがｏｆｆｓｅｔ＝０のビットとなる。ｂａｓｅａｄｄｒｅｓｓのサイズが異なれば、実際に操作されるビットも異なったものになるので、注意が必要である。

【００４４】５－２．ビットフィールド

・符号付きビットフィールド

図４に示すように太線内が対象ビットフィールドである。

0 < width≦ 32 (<<LX>> 0 < width≦64)

S:符号ビット

ｂａｓｅ＿ａｄｄｒｅｓｓのＭＳＢから、対象ビットフィールドのＭＳＢ（符号ビット）までのビットの隔たりがｏｆｆｓｅｔとなる。

ＢＦ：Ｇ命令によるメモリ上のビットフィールド操作の場合、ｏｆｆｓｅｔは任意。

ＢＦ：Ｅ命令によるメモリ上のビットフィールド操作、およびレジスタ上のビットフィールド操作の場合、ｂａｓｅ＿ａｄｄｒｅｓｓの１ワード（１ロングワード）をはみ出した部分のビットフィールドについて、動作を保証しない。

・符号なしビットフィールド

図５に示すように太線内が対象ビットフィールドである。

0 < width≦ 32 (<<LX>> 0 < width≦ 64)

ｂａｓｅ＿ａｄｄｒｅｓｓのＭＳＢから、対象ビットフィールドのＭＳＢまでのビットの隔たりがｏｆｆｓｅｔとなる。

ＢＦ：Ｇ命令によるメモリ上のビットフィールド操作の場合、ｏｆｆｓｅｔは任意。

ＢＦ：Ｅ命令によるメモリ上のビットフィールド操作、およびレジスタ上のビットフィールド操作の場合、ｂａｓｅ＿ａｄｄｒｅｓｓの１ワード（１ロングワード）をはみ出した部分のビットフィールドについて、動作を保証しない。

22

・任意長ビットフィールド

offset、width とも任意。ただしwidth ＞０。

【００４５】５－３．整数

図６、図７に整数のデータタイプを示す。

【００４６】５－４．浮動小数

浮動小数点の演算は、コプロセッサで扱う。浮動小数点の形式はＩＥＥＥ規格である。詳細は別に定める。

・単精度３２ビット浮動小数〈〈コプロセッサ〉〉

・倍精度６４ビット浮動小数〈〈コプロセッサ〉〉

・８０ビット浮動小数〈〈コプロセッサ〉〉

【００４７】５－５．１０進数

多倍長の１０進数の四則演算は、コプロセッサで扱う。本発明装置のメインプロセッサでは、以下に示すような固定長の符号なしＰＡＣＫＥＤ形式１０進数、および符号付きＰＡＣＫＥＤ形式１０進数を扱う。ただし、符号付きＰＡＣＫＥＤ形式１０進数を扱う命令は、すべて〈〈Ｌ２〉〉である。図８、図９にデータタイプをしめす。

【００４８】５－６．ストリング

図１０、図１１にストリングの場合のデータタイプを示す。

【００４９】５－７．キュー

図１２にダブルリンクでつながれた線形リストのデータタイプを示す。

【００５０】６．命令フォーマット

命令は１６ビット単位で可変長となっており、奇数バイト長の命令はない。２オペランド命令には、大きくわけて、４バイト＋拡張部の構成をもち、すべてのアドレッシングモード（Ｅａ）が利用できる一般形、および頻度の高い命令とアドレッシングモード（Ｓｈ）のみを使用できる短縮形、の２つのフォーマットがある。必要となる命令機能とコードサイズに合わせて、より適した方を選択することができる。

【００５１】本発明装置の命令フォーマットは、細かい点まで気をつければかなり多くの種類に分かれる。しかし、理解を容易にするため、ここでは本発明装置の命令フォーマットを大まかに分類して説明を行なう。命令フォーマットの詳細については、付録１０を参照のこと。フォーマット中に現われる記号の意味は次の通りである。

- オペコードの入る部分

#　リテラル、またはイミディエート値の入る部分

Ea　８ビットで指定する一般形のアドレッシングモード

Sh　６ビットで指定する短縮形のアドレッシングモード

Rn　レジスタの指定を行なう部分

フォーマットの記述は、右側がＬＳＢ側で、かつ高いアドレスになっている。（ｂｉｇ－ｅｎｄｉａｎ）

フォーマットの記述例を図１３に示す。アドレスＮとアドレスＮ＋１の２バイトを見ないと命令フォーマットが判別できないようになっているが、これは、命令が必ず１６ビット（２バイト）単位でフェッチ、デコードされ

23

ることを前提としたためである。いずれのフォーマットの場合も、各オペランドのＥａまたはＳｈの拡張部は、必ずそのＥａまたはＳｈの基本部を含むハーフワードの直後に置かれる。これは、命令により暗黙に指定されるイミディエートデータや、命令の拡張部に優先する。したがって、４バイト以上の命令では、Ｅａの拡張部によって命令のオペコードが分断される場合がある。また、付加モードなどによって、Ｅａの拡張部にさらに拡張部＊

24

＊が付く場合にも、次の命令オペコードよりもそちらの方が優先される。例えば、第一ハーフワードにＥａ１を含み、第二ハーフワードにＥａ２を含み、第三ハーフワードまである６バイト命令の場合を考える。Ｅａ１に付加モードを使用したため、普通の拡張部のほかに付加モード拡張部もつくものとする。この時、実際の命令ビットパターンは

命令の第一ハーフワード

(Ea1 の基本部を含む)

Ea1 の拡張部

Ea1 の付加モード拡張部

命令の第二ハーフワード

(Ea2 の基本部を含む)

Ea1 の拡張部

命令の第三ハーフワード

の順となる。

【００５２】なお、アラインメントの関係で１６ビットのフィールドのうちの８ビットのみを使用するケースでは、使用する８ビットは下位順（アドレスの大きい方）に詰めて置かれるものとする。これに該当するのは、オペランドサイズが８ビットで、ＥａＲ，ＳｈＲに＃ｉｍｍ＿ｄａｔａのモードを指定した場合、Ｉ－ｆｏｒｍａｔでオペランドサイズが８ビットの場合、ＢＲＡ：Ｇ，Ｂｃｃ：Ｇ，ＢＳＲ：ＧでＳＳ＝００の場合、などである。例えば、

MOV:I.B #H'12. @R0

の場合、第一バイトがＭＯＶ：Ｉ．Ｂのオペコード、第二バイトがオペコードの一部とＳｈＷ（@Ｒ０）の指定、第三バイトは０、第四バイトがＨ’１２となり、ビットパターンは図１４のようになる。この場合、１６ビットのフィールドの上位側（アドレスの小さい方）の８ビットには必ず０を入れておかなければならない。上位８ビットが０でない場合は、これによって表現されるデータがインプリメント依存の不定値になるものとする。つまり、Ｉ－ｆｏｒｍａｔ，＃ｉｍｍ＿ｄａｔａモードの場合はそのオペランドがインプリメント依存の値になり、ＢＲＡ：Ｇ，Ｂｃｃ：Ｇ，ＢＳＲ：Ｇ命令の場合は、ジャンプ先が不定となる。いずれの場合も、ＥＩＴ（例外）とはしない。

【００５３】６－１．２オペランド短縮形

【００５４】６－１－１．レジスターメモリ間（Ｓ－ｆｏｒｍａｔ，Ｌ－ｆｏｒｍａｔ）

図１５にその例を示す。Ｌ－ｆｏｒｍａｔ，Ｓ－ｆｏｒｍａｔの命令には、サイズ指定のできるもの（ＭＯＶ：Ｌ，ＭＯＶ：Ｓ，ＣＭＰ：Ｌ）とサイズ指定のできないもの（ＡＤＤ：Ｌ，ＳＵＢ：Ｌ）がある。サイズ指定のできる命令では、ＲＲ等によるサイズ指定はメモリ側のみに適用され、レジスタ側のサイズは３２ビット固定となっている。レジスタ側とメモリ側のサイズが異なる場合には、ソース側のサイズが小さい場合に符号拡張が、デスティネーション側のサイズが小さい場合に上位バイトのカットとオーバーフローのチェックが行なわれる。一方、サイズ指定のできないＡＤＤ：Ｌ，ＳＵＢ：Ｌ命令では、レジスタ側、メモリ側のオペランドサイズとも３２ビット固定である。レジスタ側のサイズを３２ビット固定としたのは、本発明装置において、「レジスタ上のデータは、できる限り３２ビット符号付き整数として扱う」という原則を設けているためである。この原則は、Ｌ－ｆｏｒｍａｔ，Ｓ－ｆｏｒｍａｔ命令のほか、ビットフィールド命令や高機能命令でレジスタ上にオペランドを置く場合にも適用される。

【００５５】６－１－２．レジスターレジスタ間（Ｒ－ｆｏｒｍａｔ）

図１６にその例を示す。

【００５６】６－１－３．リテラルーメモリ間（Ｑ－ｆｏｒｍａｔ）

図１７にその例を示す。

【００５７】６－１－４．イミディエートーメモリ間（Ｉ－ｆｏｒｍａｔ）

図１８にその例を示す。Ｉ－ｆｏｒｍａｔのイミディエート値のサイズは、デスティネーション側のオペランドのサイズと共通に８，１６，３２，６４ビットとなり、ゼロ拡張、符号拡張は行なわれない。

【００５８】６－２．１オペランド一般形（Ｇ１－ｆｏｒｍａｔ）

図１９にその例を示す。

【００５９】６－３．２オペランド一般形

ここに含まれるのは、８ビットで指定する一般形アドレッシングモードのオペランドが２つ存在する命令であ

25

る。オペランドの総数は３つ以上になる場合がある。

【００６０】６－３－１．第一オペランドはメモリ読みだし（Ｇ－ｆｏｒｍａｔ）

図２０にその例を示す。

【００６１】６－３－２．第一オペランドは８ビットイミディエート（Ｅ－ｆｏｒｍａｔ）

図２１にその例を示す。このフォーマットとイミディエートーメモリ間のフォーマット（Ｉ－ｆｏｒｍａｔ）とは機能的には似たものであるが、考え方の点では大きく違っている。Ｅ－ｆｏｒｍａｔはあくまでも２オペランド一般形（Ｇ－ｆｏｒｍａｔ）の派生形であり、ソースオペランドのサイズが８ビット固定、ディスティネーションオペランドのサイズが８／１６／３２／６４ビットから選択となっている。つまり、異種サイズ間の演算を前提とし、ｄｅｓｔのサイズに合わせて８ビットのｓｒｃがゼロ拡張または符号拡張される。一方、Ｉ－ｆｏｒｍａｔは、特にMOV，CMPで頻度の多いイミディエートのパターンを短縮形にしたものであり、ソースとディスティネーションのサイズは等しい。

【００６２】６－３－３．第一オペランドはアドレス計算（ＧＡ－ｆｏｒｍａｔ）

図２２にその例を示す。

【００６３】６－３－４．その他の２オペランド命令

図２３にその例を示す。

【００６４】６－４．ショートブランチ

図２４にその例を示す。

【００６５】６－５．その他

以上の外に図２５に示すようなものがある。

【００６６】７．アドレッシングモード

本発明装置のアドレッシングモードには、レジスタを含めて６ビットで指定する短縮形（Ｓｈ）と、８ビットで指定する一般形（Ｅａ）がある。未定義のアドレッシングモードを指定した場合や、意味的に考えて明らかにおかしなアドレッシングモードの組み合わせを指定した場合には、未定義命令を実行した場合と同じく予約命令例外（ＲＩＥ）を発生し、例外処理を起動する。これに該当するのは、ｄｅｓｔｉｎａｔｉｏｎがイミディエートモードの場合、アドレス計算の命令でイミディエートモードを使用した場合などである。

【００６７】７－１．Ｐビット

本発明装置では、毎回のメモリアクセスに対応して１ビットのオプション機能指定ビットを割り当てることができるようになっており、このビットをＰビットと呼ぶ。Ｐビットは、メモリアクセスに伴って何らかの別の意味を加えたい場合に使用するビットである。Ｐビットは、毎回のメモリアクセス毎に独立に指定する。したがって、レジスタ間接アドレッシング、アブソリュートアドレッシングなどの場合はオペランドに対応して一つのＰビットを指定するが、付加モードを使用した多段間接のアドレッシングモードでは、その段数分だけのＰビット

26

を指定することになる。Ｐビットの用途としては、タグのチェック、論理空間の切り替え、３２ビットアドレッシングと６４ビットアドレッシングの切り替えなどがあるが、これらはすべて将来の拡張用であり、現在の仕様ではＰビットはｒｅｓｅｒｖｅｄとなっている。命令フォーマットの説明では、Ｐビットの部分を’Ｐ’で表示してあるが、ここは必ず０にしておかなければならない。Ｐビットが０になっていなかった場合には、予約命令例外（ＲＩＥ）が発生する。Ｐビットに関する機能は〈〈ＬＵ〉〉仕様である。

【００６８】７－２．フォーマット中で使われる記号

Rn	レジスタ指定
P	P ビット（0でなければならない）
mem[EA]	EAで示されるアドレスのメモリ内容

以下点線で囲まれた部分は、拡張部を示す。

【００６９】７－３．レジスタ直接

アセンブラ表記：

```
Rn
```

オペランド：

```
Rn
```

フォーマット：図２６に示す。

【００７０】７－４．レジスタ間接

アセンブラ表記：

```
@Rn
```

オペランド：

```
mem[Rn]
```

フォーマット：図２７に示す。

【００７１】７－５．レジスタ相対間接

アセンブラ表記：

```
@(disp,Rn)
@(disp:16,Rn)
@(disp:32,Rn)
```

オペランド：

```
mem[disp + Rn]
```

フォーマット：図２８に示す。

なおｄｉｓｐは符号付きとして扱う。

【００７２】７－６．イミディエート

アセンブラ表記：

```
#imm__data
```

オペランド：

```
imm __data
```

フォーマット：図２９に示す。

なおｉｍｍ__ｄａｔａのサイズは、オペランドサイズとして命令中で指定される。

【００７３】７－７．アブソリュート

アセンブラ表記：

27

@abs
@abs:16
@abs:32
@abs:64 <<LX>>

オペランド：

mem[abs]

フォーマット：図３０に示す。
なお３２ビットアドレッシングの時は、ａｂｓ：１６で指定したアドレスは３２ビットに符号拡張される。また、６４ビットアドレッシングの時は、ａｂｓ：１６，ａｂｓ：３２で指定したアドレスは６４ビットに符号拡張される。

【００７４】７－８．ＰＣ相対間接
アセンブラ表記：

@(disp.PC)
@(disp:16.PC)
@(disp:32.PC)

オペランド：

mem[disp + PC]

フォーマット：図３１に示す。
ＰＣ相対間接モードにおいて参照されるＰＣの値は、そのオペランドを含む命令の先頭アドレスである。したかって、例えば無限ループは

JMP @(0.PC)

という命令によって実現される。付加モードにおいてＰＣの値が参照される場合にも、同じように命令先頭のアドレスをＰＣ相対の基準値として使用する。

【００７５】７－９．スタックポップ
アセンブラ表記：

@SP+

オペランド：

mem[SP]
SPをインクリメント

フォーマット：図３２に示す。
＠ＳＰ＋のモードでは、オペランドサイズだけＳＰをインクリメントする。例えば、本発明装置６４で６４ビットデータを扱う時には、ＳＰが＋８だけ更新される。Ｂ，Ｈのサイズのオペランドに対する＠ＳＰ＋の指定も可能であり、それぞれＳＰが＋１，＋２だけ更新される。ただし、スタックのアラインメントがくずれて速度低下の原因になるため、使用上は注意した方がよい。オペランドに対して＠ＳＰ＋のモードが意味を持たないものに対しては、予約命令例外（ＲＩＥ）を発生する。具体的に予約命令例外（ＲＩＥ）となるのは、ｗｒｉｔｅオペランド、ｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅオペランドに対する＠ＳＰ＋である。

【００７６】７－１０．スタックプッシュ

28

アセンブラ表記：

@-SP

オペランド：

SPをデクリメント
mem[SP]

フォーマット：図３３に示す。
＠－ＳＰのモードでは、オペランドサイズだけＳＰをデクリメントする。例えば、本発明装置６４で６４ビットデータを扱う時には、ＳＰが－８だけ更新される。Ｂ，Ｈのサイズのオペランドに対する＠－ＳＰの指定も可能であり、それぞれＳＰが－１，－２だけ更新される。ただし、スタックのアラインメントがくずれて速度低下の原因になるため、使用上は注意した方がよい。オペランドに対して＠－ＳＰのモードが意味を持たないものに対しては、予約命令例外（ＲＩＥ）を発生する。具体的に予約命令例外（ＲＩＥ）となるのは、ｒｅａｄオペランド、ｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅオペランドに対する＠－ＳＰである。

【００７７】７－１１．レジスタ相対付加モード
オペランド：

Rn ==> tmp
付加モード処理

フォーマット：図３４に示す。付加モードについては、後の章でまとめて説明する。

【００７８】７－１２．ＰＣ相対付加モード
オペランド：

PC ==> tmp
付加モード処理

フォーマット：図３５に示す。

【００７９】７－１３．絶対付加モード
オペランド：

0 ==> tmp
付加モード処理

フォーマット：図３６に示す。

【００８０】７－１４．ＦＰ相対間接
アセンブラ表記：

@(disp.FP)
@(disp:4.FP)

オペランド：

mem[d4 * 4 + FP]
(disp = d4 * 4)

フォーマット：図３７に示す。
ｄ４は符号付きとして扱い、オペランドのサイズとは関係なく必ずｄ４を４倍して使用する。したがって、この

29

モードにより（ＦＰ－８＊４）から（ＦＰ＋７＊４）までの４の倍数のメモリアドレスが参照可能である。アセンブラで記述する場合には、ディスプレースメントとして４倍した値の方を書く。このアドレッシングモードは〈〈Ｌ２〉〉である。本発明装置ではＦＰ相対間接モードは実装しないので、このアドレッシングモードが指定された場合は、予約命令例外（ＲＩＥ）となる。このアドレッシングモードは短縮形で利用できないので、例えば、

```
MOV     @(disp.FP).R1
```

といった場合に、

```
MOV:G.W @(disp:4.FP).R1
MOV:L.W @(disp:16.FP).R1
```

がともに４バイトとなり、コードの選択に曖昧さが生じるという問題点がある。このモードが〈〈Ｌ２〉〉となっているのは、このような理由による。このモードは、本発明装置６４になって短縮形の割合が減った時に、有効に利用することを狙ったものである。@（ｄ４：４，ＦＰ）、@（ｄ４：４，ＳＰ）のモードでは、オペランドサイズにかかわらずｄ４を４倍して使用するため、８ビット、１６ビット、３２ビットのローカル変数をスタックフレーム上に混在した場合に@（ｄ４：４，ＦＰ）、@（ｄ４：４，ＳＰ）のモードを利用しようとすると、本発明装置がｂｉｇ－ｅｎｄｉａｎである関係上、各変数のＭＳＢ側をワード境界に合わせて配置する必要がある。これによって特に問題が起きるわけではないが、注意が必要である。@（ｄ４：４，ＦＰ），@（ｄ４：４，ＳＰ）モードを利用するためのローカル変数配置の例を図３８に示す。

【００８１】７－１５．ＳＰ相対間接

アセンブラ表記：

```
@(disp.SP)
@(disp:4.SP)
```

オペランド：

```
mem[d4 * 4 + SP]
(disp = d4 * 4)
```

基本フォーマット：図４０に示す。

30

フォーマット：図３９に示す。

ｄ４は符号付きとして扱い、オペランドのサイズとは関係なく必ずｄ４を４倍して使用する。ただし、ｄ４が負の値であった場合の動作は規定されていない。したがって、このモードにより（ＳＰ）から（ＳＰ＋７＊４）までの４の倍数のメモリアドレスが参照可能である。アセンブラで記述する場合には、ディスプレースメントとして４倍した値の方を書く。このアドレッシングモードは〈〈Ｌ２〉〉である。本発明装置ではＦＰ相対間接モードは実装しないので、このアドレッシングモードが指定された場合は、予約命令例外（ＲＩＥ）となる。このモードも、@（ｄｉｓｐ：４，ＦＰ）と同様に、本発明装置６４になって短縮形の割合が減った時に、有効に利用することを狙ったものである。

【００８２】７－１６．付加モードのフォーマット

複雑なアドレッシングも、基本的には加算と間接参照の組み合わせに分解することができる。したがって、加算と間接参照のオペレーションをアドレッシングのプリミティブとして与えておき、それを任意に組み合わせることができれば、どんな複雑なアドレッシングモードをも実現することができる。付加モードはこのような考え方にたったアドレッシングモードである。複雑なアドレッシングモードは、モジュール間のデータ参照やＡＩ言語の処理系に特に有用である。ただし、本発明装置ではメモリ間接アドレッシングモードが多用された場合処理速度が低下する場合があるので、メモリ間接アドレッシングモードの使用に際しては十分な注意が必要である。付加モードの指定は、１６ビットを単位としており、これを任意回繰り返す。１段の付加モードにより、

定数(displacement)の加算

インデクスレジスタのスケーリングと加算

スケーリングは×１、×２、×４、×８

メモリの間接参照

を行なう。Ｎ段の付加モードにより、Ｎ＋１段までの間接参照ができる。

基本的な付加モードの処理：

```
tmp + Rx * scale + d4 * 4 ==> tmp          I=0,D=0 の時
tmp + Rx * scale + dispx  ==> tmp          I=0,D=1 の時
mem[tmp + Rx * scale + d4 * 4] ==> tmp     I=1,D=0 の時
mem[tmp + Rx * scale + dispx] ==> tmp      I=1,D=1 の時
```

31　　32

EI=00　間接参照なし、付加モード継続

tmp + disp + Rx * Scale ==> tmp

EI=01　間接参照あり、付加モード継続

mem[tmp + disp +Rx * Scale] ==> tmp

EI=10　間接参照なし、付加モード終了

tmp + disp + Rx * Scale ==> address of operand

EI=11　間接参照あり、付加モード終了

mem[tmp + disp +Rx * Scale] ==> address ofoperan

M=0　　<Rx> をインデクスとして使用

M=1　　特殊なインデクス

<Rx>=0　インデクスを加算しない

(Rx=0)

<Rx>=1　PC をインデクスRxとして使用

(Rx=PC)

<Rx>=2 ～ reserved

D=0　　付加モード中の４ビットのd4を４倍してdispとし、これを加算する。d4は符号付きとして扱い、オペランドのサイズとは関係なく必ずd4を４倍して使用する。

D=1　　付加モードの拡張部で指定されたdispx（16/32/64ビット）をdispとし、これを加算する。拡張部のサイズはd4フィールドで指定する。

d4=0001　dispx は16ビット

d4=0010　dispx は32ビット

d4=0011　dispx は64ビット　<<LX>>

XX　　インデクスのスケール(scale = 1/2/4/8)

S　　インデクスレジスタのサイズ

S=0　<Rx> は32ビットを符号拡張

S=1　<Rx> は64ビット　<<LX>>

P　　Ｐビット <<LU>>

・Ｐビットは付加モードの各段に入る。Ｐビットは「すべてのメモリ参照で独立に指定できるビット」となっている。

・間接参照をする場合としない場合を選択できる。間接参照しない段は、多段のベースレジスタ、インデクスレジスタの加算に用いる。（ｍｅｍ［Ｒ１＋Ｒ２＋Ｒ３］など）これは、ユーザレベルでｒｅｌｏｃａｔｉｏｎ　ｂａｓｅ　ｒｅｇｉｓｔｅｒなどを導入したい時に使用することがある。

・インデクスレジスタのサイズ

６４ビットアドレス使用時でも３２ビットデータがかなりの頻度で出てくると予想されるため、付加モードの各段で３２／６４のサイズ切り替えができるようになっている。

・レジスタ相対間接の@（ｄｉｓｐ：６４，Ｒｎ）やメモリ間接のアドレッシングモードも付加モードを使用して実現する。

・ＰＣに対して×２、×４、×８のスケーリングを行なった場合には、その段の処理終了後の中間値（ｔｍｐ）として、インプリメントに依存した不定値が入る。この付加モードによって得られる実効アドレスは予測できない値となるが、例外は発生しない。マニュアル等では、ＰＣに対する×２、×４、×８のスケーリングの指定は行なわないように、注意しておく必要がある。

フォーマットのバリエーション：図４１，図４２に示す。

33

【００８３】７－１７．付加モード仕様のレベル
付加モードの利用方法としては、普通の間接参照、オブジェクトコードのモジュール化のための外部変数のテーブル参照、ＡＩ向け命令の実行などがあり、このうち、ＡＩ向けの用途ではかなり多くの段数の間接参照を使うことがあるが、普通の用途では、３～４段までの間接参照で十分なことが多い。任意段数の付加モードが利用できれば、コンパイラの中で段数による場合分けが不要になるので、コンパイラの負担が軽減されるというメリットがある。多段の間接参照の頻度が非常に少ないとしても、コンパイラとしては必ず正しいコードを発生できなければならないからである。しかし、インプリメントの方から考えると、任意の段数を許して実行中の割り込みを受け付けるようにするのはかなり重くなるため、全体のバランスとしてある程度の段数制限をするのはやむを得ない。そこで、本発明装置としては４段（付加モード、基本フォーマットの４つ分）までの付加モードが利用できるものを〈〈Ｌ１〉〉仕様とし、任意段数の付加モードのインプリメントは〈〈Ｌ２〉〉仕様としてクラス分けする。〈〈Ｌ１〉〉仕様でも、５回のメモリ間接参照まで可能である。５段（５ハーフワード）以上の付加モードに対しては、予約命令例外（ＲＩＥ）が起動される。ただし、フォーマット上は任意の段数が可能になっているので、将来はそのままのフォーマットで段数を拡張することができる。本発明装置では任意段数の付加モードを許す。ただし、本発明装置では付加モードをもちいてメモリ間接アドレッシングを多用した場合、処理速度が低下することがあるので注意を要する。特に第２オペランドで多段の付加モードが用いられた場合、付加モード処理中は割り込みが受け付けられない場合があるので注意を要する。また、本発明装置３２でも浮動小数点を扱うことを考え、’×８’のスケーリングをインプリメントする。’×８’のスケーリングは〈〈ＬＸ〉〉仕様ではなく〈〈Ｌ１〉〉仕様である。
【００８４】８．インプリメント関連事項
【００８５】８－１．仮想記憶のサポート
（本発明装置では仮想記憶のサポートは行わない。）
仮想記憶を実現するため、命令の実行途中で発生したページフォールトに対してうまく回復処理を行なう必要がある。本発明装置では、原則として命令再実行方式を採用する。命令再実行方式でページフォールトが起こった場合には、それまでに変更したレジスタ類をプロセッサがすべてもとに戻してから、ページインの処理ルーチンを起動する。したがって、処理再開後に命令の始めより再実行しても矛盾は生じない。命令再実行方式では、原則として実行途中の状態を保持する必要がないので、実現機構は比較的簡単である。また、本発明装置では、命令再実行のことを考慮し、処理の途中で副作用を残す命令やアドレッシングモード（オートインクリメントなど）を極力避けるようにしている。ただし、ページフォールトからの再実行では余分なメモリアクセスが起こることがあり、ＯＳで入出力装置を操作する場合などには注意が必要である。例えば、一般形の命令において、第一オペランドでＩ／Ｏのリードを行ない、第二オペランドでページフォールトを起こした場合、命令の再実行でもう一度Ｉ／Ｏのリードを行なうため、Ｉ／Ｏの種類によっては矛盾を起こす。したがって、リードによって副作用のあるＩ／Ｏをアクセスする場合は、その命令のもう一方のオペランドでページフォールトを起こさないように注意し、マニュアルにも明記する必要がある。具体的には、もう一方のオペランドが必ずレジスタか常駐ページであればよい。また、ＭＯＶ命令などでソースオペランドとデスティネーションオペランドが一部重なっていた場合、単なる再実行では矛盾を生じることがあるので、この点に対する注意も必要である。
例：２バイトのデータを１バイトずらす。デスティネーションがページ境界にまたがる。図４３で、ＭＯＶ．Ｈ命令により［Ｎ－２：Ｎ－１］を［Ｎ－１：Ｎ］に移す場合、デスティネーションの書き込みバスサイクルは２回に分かれる。まず［Ｎ－２］のデータが［Ｎ－１］に書かれ、次に［元のＮ－１］が［Ｎ］に書かれるものとする。［Ｎ－１］への書き込みの際にｐａｇｅＭ－１がフォールトを起こすと、ページイン後［Ｎ－２：Ｎ－１］－－〉［Ｎ－１：Ｎ］を再試行しようとしても、Ｎ－１の内容が既に書き変わっているので、矛盾を起こす。さらに、ＬＤＭのような複数のデータ転送を行なう命令でも、転送元と転送先が重なっていた場合に、命令の再実行で矛盾が生じないようにする必要がある。例えば、

```
LDM     @R6.(R6-R10)
```

の場合、Ｒ６，Ｒ７をロードした後でＲ８を読んだ時にページフォールトが起きると、再実行した時に既にＲ６が書き換わっており、本当に命令の最初から実行すると、矛盾を生じる。これを避けるためには、以下のような対策をとる必要がある。
・命令の最初でページフォールトが起きないことを確認する。
・ページフォールト時に転送中のアドレスを示すテンポラリ値をスタックにセーブする。（一種の命令継続実行方式）
・Ｒ６の初期値を記憶しておき、ページフォールト発生時にはこれをもとの値に戻す。ＳＴＭやその他の命令についても同様である。なお、命令の再実行を矛盾なく行なうため、ＬＤＭ，ＳＴＭ，ＬＤＣＴＸでは付加モードを禁止している。また、ＥＮＴＥＲ，ＥＸＩＴ，ＪＲＮＧでは、メモリアクセスを伴うようなアドレッシングモードをすべて禁止している。
【００８６】８－２．プログラムによる命令の書き換えストアドプログラム方式の計算機では、一般に、これから自分の実行する命令プログラム自体をプログラムによ

35

って書き換えることが可能である。しかし、命令のプリフェッチや命令キャッシュなどを持つ最近の高性能プロセッサでは、プログラムで命令を書き換える場合の動作を保証しようとすると、ハードウエアの負担が極めて大きくなる。また、この機能は必要性が少なく、ソフトウエアの教育上も好ましくない。したがって、本発明装置では、ソフトウエアによってこれから実行する命令コードの書き換えを行なうことは原則として禁止し、そのような場合には動作を保証しないものとしている。ただし、ＯＳからユーザまで含めたシステム全体の動作を見ると、どこかでプログラムのロード～実行といった流れを含んでいるため、すべての場合にわたって「動作を保証しない」とするわけにはいかない。また、特殊な用途では、ユーザプログラムで命令コードを生成し、それを実行したいという場合もある。したがって、何らかの条件が満たされた時には、ソフトウエアによって書き換えられた命令コードの実行動作を保証する必要がある。そこで、本発明装置では、命令コードを書き換えたということをプロセッサに知らせる命令ＰＩＢを用意し、この命令を実行することにより、以後、書き換えられた命令コードの実行動作を保証することにしている。この命令は、これから実行すべき命令コードが、以前（リセット時あるいは前回のＰＩＢ命令実行時）から変更されている可能性があるということを、プロセッサに通知するために使用する。インプリメント上は、この命令によってパイプライン、命令キュー、命令キャッシュのパージを行なうことになる。

【００８７】９．ＥＩＴ処理

本発明装置では、プログラムの実行の流れとは非同期に行なわれる処理を総称して、ＥＩＴ処理と呼んでいる。ＥＩＴ処理は、通常、例外処理や割り込み処理と呼ばれているものである。ＥＩＴ処理には、次のようなものが含まれる。

36

・内部割り込み（リング間コール、トラップ）

システムコール発行などの際に、プログラマが意識して発生させる。その時に実行中のコンテキストとは関連がある。

・例外割り込み（例外）

一般の命令の実行中に、何らかのエラーが起った場合に発生する。その時に実行中のコンテキストとは関連がある。

・外部割り込み（割り込み）

外部からのハードウエア的な信号により発生する。その時に実行中のコンテキストとは全く関連がない。ＥＩＴとはＥｘｃｅｐｔｉｏｎ（例外割り込み）、Ｉｎｔｅｒｒｕｐｔ（外部割り込み）、Ｔｒａｐ（内部割り込み）の頭文字を合わせた名称である。ＥＩＴ処理に関する詳細は付録９を参照のこと。

【００８８】１０．ＰＳＷの構成

本発明装置のＰＳＷ（Ｐｒｏｃｅｓｓｏｒ　Ｓｔａｔｕｓ　Ｗｏｒｄ）は３２ビットである。ＰＳＷの下位１６ビット（ＰＳＨ－Ｐｒｏｃｅｓｓｏｒ　Ｓｔａｔｕｓ　Ｈａｌｆｗｏｒｄ）はユーザプログラム用であり、ユーザプロセスから自由に操作可能である。ＰＳＷの上位１６ビット（ＰＳＳ－Ｐｒｏｃｅｓｓｏｒ　Ｓｔａｔｕｓ　ｈａｌｆｗｏｒｄ　ｆｏｒＳｙｓｔｅｍ）はシステム用であり、ユーザプログラム（リング３）からは操作できない。ＰＳＨのうち、上位８ビットは各種モードの設定を行なう部分であり、ＰＳＭ（Ｐｒｏｃｅｓｓｏｒ　Ｓｔａｔｕｓ　ｂｙｔｅ　ｆｏｒ　Ｍｏｄｅ）と呼ぶ。また、ＰＳＨの下位８ビットは各種ステータスや演算結果の表示を行なう部分であり、ＰＳＢ（Ｐｒｏｃｅｓｓｏｒ　Ｓｔａｔｕｓ　Ｂｙｔｅ）と呼ぶ。図４４に示す。

【００８９】１０－１．ＰＳＳの構成

図４５に示す。

37 38

'0' にreserved
'1' を書き込もうとした場合には、予約機能例外(RFE)が発生する。

SM.RNG = 000 ring0 で外部割り込みスタックポインタ(SPI) 使用
SM.RNG = 001 reserved
SM.RNG = 010 reserved
SM.RNG = 011 reserved
SM.RNG = 100 ring0 でリング0 用スタックポインタ(SP0) 使用
SM.RNG = 101 reserved (ring1 用)
SM.RNG = 110 reserved (ring2 用)
SM.RNG = 111 ring3 でリング3 用スタックポインタ(SP3) 使用
SM.RNGは<<LA>>

XA = 0 32ビットコンテキスト

XA = 1 64ビットコンテキスト
<<LX>>

AT = 00 アドレス変換なし
AT = 01 アドレス変換あり（本発明装置標準のMMU仕様）
AT = 10 アドレス変換なし、アドレスによるメモリ保護
(<<L1R>>)

AT = 11 reserved
(Address Translation mode)
DB = 0 デバッグ中でないコンテキスト
DB = 1 デバッグ中コンテキスト

IMASK 外部割り込み、DI(Delayed Interrupt) を禁止する割り込み優先度
IMASK = 0000 NMI （優先度0 のマスク不能割り込み）のみ受け付け
IMASK = 0001 優先度1 までマスク（結果的にNMI のみ受け付けとなる）
IMASK = 0010 優先度２までマスク
... IMASK の示す割り込みより優先度の高い割り込みのみ受け付ける
IMASK = 1110 優先度14までマスク
IMASK = 1111 マスクしない

・本発明装置では、〈〈ＬＡ〉〉仕様として４レベルの 50 リング保護によるメモリ管理を行なう（付録参照）。本

39

発明装置では２レベルのリング保護によるメモリ管理を行なう。ＲＮＧフィールドは、現在プロセッサがどのリングにいるかという状態を示すものである。リング保護を行なわない場合にも、例えばスーパバイザ、ユーザモードの切り換え用にこのフィールドを使用する。

・ＸＡビットは、本発明装置３２ではｒｅｓｅｒｖｅｄであり、１を書き込もうとすると例外が発生する。

・トレースなどデバッグ関係の情報については、その詳細まで統一するのは難しいため、別の制御レジスタ（ＤＣＲ－Ｄｅｂｕｇ　Ｃｏｎｔｒｏｌ　Ｒｅｇｉｓｔｅｒ）に分離している。ただし、デバッグ中がどうかを示す情報のみＤＢとしてＰＳＷに入れる。

・本発明装置の外部割り込みは、低い優先度の方が大きな数字になる。外部割り込みの優先度は、０～６の７レベルであり、優先度０はマスク不能割り込み（ＮＭＩ）である。

・キャッシュやMMUの制御情報は完全な統一が難しい＊ため、ＰＳＷとは分離している。

40

・ＡＴ（アドレス変換指定フィールド）をＰＳＷに入れたことによって、コンテキスト毎にアドレス変換やメモリ保護の方法を変えたり、ＥＩＴ処理ハンドラ実行中のみ一時的にアドレス変換を止めたりすることが可能になっている。

なお、ＬＤＣ，ＲＥＩＴ，ＬＤＣＴＸ，ＥＩＴ起動などによって、ＰＳＷ中のＡＴ（アドレス変換ビット）が００から０１に変更された場合には、ＴＬＢやキャッシュのパージが自動的に行なわれ、ＴＬＢや論理キャッシュの整合性が保証されるものとする。また、ＡＴが０１から００に変更された場合にも、キャッシュ（この場合は論理キャッシュ兼物理キャッシュ）の整合性が保証されるものとする。

【００９０】１０－２．ＰＳＨの構成

図４６に示す。

-	'0' にreserved '1' を書き込もうとした場合には、予約機能例外 (RFB)が発生する。
PRNG	このリングに入る一つ前の状態のリング番号 PRNGは<<LA>>
P	P-bit Error Flag　<<LU>> P-bit 機能の関連でエラーが起きたときにセットされ、他の場合にはクリアされる。 現在は0 にreserved
F	General Flag 高機能命令の終了要因の判定などに用いる。
X	Extension Flag 多倍長計算用の桁上がりなどを示す。
V	Overflow Flag オーバーフローが発生したことを示す。
L	Lower Flag 比較命令などにおいて、第一オペランドの方が小さいことを示す。（符号付き演算、符号なし演算とも）
M	MSB Flag 演算結果のMSB が1 であることを示す。
Z	Zero Flag 演算結果が0 になったことを示す。

・ＰＲＮＧフィールドで「一つ前のリング」とは、「一つ外側のリング」あるいは、「そのリングにサービスを依頼したリング」を表わすものである。したがって、ＥＩＴ発生時のＰＲＮＧの変化は、

PS　　W<RNG> ==> PSW<PRNG>

リターン時（ＲＥＩＴ命令）でのＰＲＮＧの変化は、

スタック ==> PSW(RNG, PRNG を含む)

となる。リターン時はＲＮＧよりコピーするのではなく、必ずスタックより復帰する必要がある。常にＲＮＧ≦ＰＲＮＧが成立する。ＰＲＮＧは、ＡＣＳ命令などでの参照を目的としたもので、実際のリング遷移はあくまでもＲＮＧの情報を使用する。

・本発明装置以外のプロセッサでは、比較～条件ジャンプ、といった命令の流れをとる場合に、符号付きと符号

41

なしの区別を比較命令ではなく条件ジャンプ命令で行なうのが普通である。例えば、符号なし整数の比較を

```
CMP    src1, src2
BLTS   next
   Branch Lower Than(Signed)
```

で、符号付き整数の比較を

```
CMP    src1, src2
BLTU   next
   Branch Lower Than(Unsigned)
```

で行なう。したがって、フラッグの表現する情報として、大小の区別のほかに、符号付きと符号なしの区別も必要である。しかし、本発明装置では、符号付きと符号なしの区別がＣＭＰ命令、ＣＭＰＵ命令といったように命令別になっており、条件ジャンプ命令は符号付きも符号なしも共通である。したがって、フラッグ構成を簡単にすることができる。

・通常のプロセッサで使用するＣａｒｒｙ　Ｆｌａｇは、符号なし整数の大小関係を表わすという意味と、多倍長演算の桁上がりを表わすという意味がある。しかし、後者に関しては本発明装置ではＸ＿ｆｌａｇを使用するため、ＣａｒｒｙＦｌａｇは整数の大小関係を表わすという意味でのみ用いられる。したがって、ＴＲＯＮ　ＣＨＩＰではこのフラッグを大小関係を表わすフラッグであると定義し、名前をＬ＿ｆｌａｇ（Ｌｏｗｅｒ　Ｆｌａｇ）としている。このフラッグは、符号なし演算の場合には従来のＣａｒｒｙ　Ｆｌａｇと同じ振る舞いをするが、符号付き演算の場合には従来のＣａｒｒｙ　Ｆｌａｇとは異なり、オーバーフローまで考慮した真の大小関係を表現する。

・そのほか、ストリング命令やキューの命令の終了条件を示すためのＦ＿ｆｌａｇ（Ｇｅｎｅｒａｌ　Ｆｌａｇ）とＰビットのエラーを表現するためのＰ＿ｆｌａｇ（Ｐ－ｂｉｔ　Ｅｒｒｏｒ　Ｆｌａｇ）を設ける。Ｐ＿ｆｌａｇは、現在の仕様では’０’にｒｅｓｅｒｖｅｄとなっている。

・通常のプロセッサでは、シフト命令ではみだしたビットを入れるためにＣａｒｒｙ　Ｆｌａｇを用いているが、本発明装置ではＣａｒｒｙ　Ｆｌａｇの代わりにＬ＿ｆｌａｇを実装しているため、はみだしたビットはＸ＿ｆｌａｇに入れることにする。

【００９１】１０－３．フラッグの変化

例：　@dest.B = 1 の時

```
SUB    #H'101.W,@dest.B   ==>演算結果1 - H'101
          は0 でないが、destが0 になるので Z_flagはセッ
          トされる。
CMP    #H'101.W,@dest.B   ==> 演算結果1 - H'101
          は0 でないため、 z_flagはクリアされる。
```

42

加減算命令、比較命令、論理演算命令は２オペランド命令であり、

```
dest .op. src --> dest
```

の形をとる。ｄｅｓｔとｓｒｃのサイズが異なる場合には、小さいサイズの方が大きいサイズに合わせて符号拡張（ＡＤＤＵ，ＳＵＢＵ，ＣＭＰＵではゼロ拡張）された上で演算され、演算結果がｄｅｓｔのサイズに変換されてからｄｅｓｔに格納される。ＣＭＰ，ＣＭＰＵ，ＳＵＢ，ＳＵＢＵの場合、Ｌ＿ｆｌａｇは、前の演算で第一オペランドの方が値が小さかったことを示す。符号なし演算ＣＭＰＵ，ＳＵＢＵの場合には、Ｌ＿ｆｌａｇは通常のプロセッサのＣａｒｒｙ（Ｂｏｒｒｏｗ）Ｆｌａｇと同じ意味になる。符号付き演算の場合には、Ｌ＿ｆｌａｇは単なるＭ＿ｆｌａｇのコピーとは異なり、オーバーフローまで含めた真の大小関係を表現する。ＡＤＤ命令の場合には、Ｌ＿ｆｌａｇは結果が負であることを示す。これも、単なるＭ＿ｆｌａｇのコピーとは異なり、オーバーフローまで含めた真の正負を示す。ＡＤＤＵの場合には、結果が必ず正になるため、Ｌ＿ｆｌａｇは０となる。Ｖ＿ｆｌａｇは、演算の結果がｄｅｓｔで指定されたサイズでは表現できなかったということを示す。つまり、演算結果がｄｅｓｔのサイズの符号付き整数（ＡＤＤＵ，ＳＵＢＵでは符号なし整数）で表現できない時に、Ｖ＿ｆｌａｇがセットされる。ＣＭＰ，ＣＭＰＵ命令では、Ｖ＿ｆｌａｇは不変である。Ｘ＿ｆｌａｇは、多倍長の演算を行なう場合に、桁上がりの状態を保持するために使用する。符号付き演算でも符号なし演算の時と同じような変化をする。これは、通常のプロセッサのＣａｒｒｙ　Ｆｌａｇとほぼ同じ意味であるが、Ｘ＿ｆｌａｇを変化させる命令は加減算命令やシフト命令などに限られている。ＣＭＰ命令とＳＵＢ命令、およびＣＭＰＵ命令とＳＵＢＵ命令のＬ＿ｆｌａｇの変化は全く同じである。Ｘ＿ｆｌａｇは、ＳＵＢ，ＳＵＢＵ，ＳＵＢＸ命令では変化するが、ＣＭＰ，ＣＭＰＵ命令では変化しない。ＭＯＶ，ＭＯＶＵ，ＡＤＤ，ＡＤＤＵ，ＡＤＤＸ，ＳＵＢ，ＳＵＢＵ，ＳＵＢＸ命令の場合、Ｍ＿ｆｌａｇとＺ＿ｆｌａｇは、演算結果をｄｅｓｔのサイズに変換した後の値を基準にして変化する。したがって、ｓｒｃのサイズよりもｄｅｓｔのサイズの方が小さい時は、演算結果が０でなくてもＺ＿ｆｌａｇがセットされることがありうる。一方、ＣＭＰ，ＣＭＰＵ命令の場合のＺ＿ｆｌａｇは、演算結果そのものの値を基準にして変化し、ｄｅｓｔのサイズには関係しない。

43

ＡＤＤＸ，ＳＵＢＸ命令のフラッグの変化は、多少変則的になっている。これは、符号なし整数の拡張演算にも符号付き整数の拡張演算にも対処するためである。この＊

44

＊場合、条件ジャンプ命令のニモニックとの対応がうまくとれなくなるが、拡張演算自体が頻度も少なく、変則的な面を持っているので、やむを得ない。

L__flag　符号付き演算としての大小関係(SUBX)、正負(ADDX)を示す。

V__flag　符号付き演算としてのオーバーフローを示す。

X__flag　ADDXの場合はdest + src + X__flagの演算におけるdestのサイズからの桁上がり、SUBXの場合は、dest - src - X__flagの演算におけるdestのサイズからの桁下がりを表わす。ただし、いずれの場合も、src のサイズがdestのサイズよりも小さい場合には、src が符号拡張される。SUBXにおいて、src とdestのサイズが等しい場合には、結果的に、X__flagが符号なしデータとしての比較結果を表わすことになる。

ＡＤＤＸ，ＳＵＢＸで異種サイズ間の演算を行なう場合には、サイズの短い方が符号拡張される。しかし、符号拡張後の値を符号付きの数とみて演算するか、符号なしの数とみて演算するかはフラッグによって異なる。ＭＯＶ命令、ＭＯＶＵ命令および論理演算命令では、Ｘ＿ｆｌａｇ，Ｌ＿ｆｌａｇは変化しない。論理演算命令では、Ｖ＿ｆｌａｇも変化しない。各命令に対応したフラッグの変化は、命令セットの説明の中に示されている。’☆’は要注意個所である。

【００９２】１１．命令セットの記述について

【００９３】１１－１．記述形式の概要

〔ニモニック〕その命令の名前（ニモニック）を示す。

〔命令の機能〕その命令の機能の概要を示す。

〔命令オプション〕その命令で使用できる命令オプションの種類を示す。命令オプションは、命令の機能の細かい点を変更するために用いるものであり、アセンブラ表記では’／ｘｘｘ’により記述する。

〔命令ビットパターンとアセンブラ表記〕命令のビットパターン、そのアセンブラ表記、使用できるサイズの種類などを示す。本発明装置では、一つの命令ニモニックに対して一般形や短縮形といった複数の命令フォーマットが存在する場合があり、それぞれ使用できるアドレッシングモードやサイズが異なっている。この項では、命令フォーマット別にそういった内容を明らかにする。

〔フラッグ変化〕命令実行後のステータスフラッグ（ＰＳＢ）の変化を示す。

〔解説〕その命令の機能を解説する。なお、説明の中で現れるアセンブラニモニックの詳細については、巻末の付録を参照のこと。

【００９４】１１－２．命令ビットパターンとアセンブラ表記

命令ビットパターンとアセンブラ表記の部分は、フォーマット別ニモニック、オペランド名、オペランドフィールド名、命令ビットパターンから成る。

記述例：図４７に示す。

ＡＮＤ：Ｇ－－フォーマット別ニモニック

説明を行なう命令ビットパターンのフォーマット別ニモニック（付録参照）を示す。

ｓｒｃ，ｄｅｓｔ－－オペランド名

その命令の機能を説明するために使用する変数である。この変数は、「命令の機能」「解説」で参照される。ここで記述されたオペランドの順番が、そのままアセンブラにおけるオペランドの順番になる。

ＥａＲ，ＥａＭ－－オペランドフィールド名

オペランドフィールド名は、ビットパターンとの対応、使用できるオペランドサイズやアドレッシングモード、メモリアクセス方法、その他の制約事項などの情報をまとめて表わすものである。オペランドフィールド名を表わす文字とその意味との間には一定の原則を設けておき、いろいろな意味を簡潔に表現できるようにしている。

枠でかこまれた部分－－命令ビットパターン

命令ビットパターン中では、オペランドフィールドやサイズ指定フィールドの位置、命令のオペコードなどを示す。’＊’で示されるビットは、ｄｏｎ’ｔｃａｒｅのビットである。このビットの０／１は、命令デコードには影響しない。’－’，’＋’，’＝’，’＃’で示されるビットは、現在のところ、命令機能やオペランドの区別には使用されていないビットである。ただし、ユーザプログラムでは’－’，’＝’の部分には０を、’＋’，’＃’の部分には１を入れておかなければならない。’－’のビットが０でない場合や’＋’のビットが１でない場合は、予約命令例外（ＲＦＥ）となる。’＝’のビットが０でない場合や’＃’のビットが１でない場合は、単に無視される。つまり、ハード的には’＊’，’＝’，’＃’は同等の意味を持つ。しかし、将

45

来の拡張のために、ユーザ向けのマニュアルには'=', '#'を'0', '1'としておくように明記しておかなければならない。

【００９５】１１－３．フィールド名

命令ビットパターン中には、オペランドフィールドのほか、オプションフィールド、サイズ指定フィールドがある。本発明装置で使用しているオプションフィールド名、サイズ指定フィールド名には、次のようなものがある。

・サイズ指定フィールド名

RR	readアクセスを行なうオペランドのサイズ指定
WW	write アクセスを行なうオペランドのサイズ指定
MM	read-modify-write アクセスを行なうオペランドのサイズ指定
BB	ビット操作命令でのメモリアクセスサイズ
XX	上記以外の一般的なサイズ指定 特にレジスタサイズの指定を行なう場合
SS	上記以外の一般的なサイズ指定 ディスプレースメントのサイズ、CMP の第二オペランド、暗黙でオペランドを指定するストリング命令、暗黙でスタックを指定するMOVA:U命令などに使用 必ず同じ文字（大文字）を反復する。ただし、32ビットと64ビットの指定しかできない場合には、このうちの一文字のみを使う。

・オプションフィールド名

命令オプションの指定を行なうオプションビットの名前としては、主として小文字を使う。（Ｐビット関係を除く）

※オプションフィールド名には、以下に示すようなものがある。いずれの場合にも、最初に記述した方（オプション値が０，００．．の方）がアセンブラでのデフォルトになる。

cccc	Bcc.TRAP/cc での条件指定
eeee	ストリング命令、QSCH命令での終了条件指定
P,Q..	P ビット指定（Q.. は、P ビットの必要なオペランドが複数の場合）
b	/F=0./B=1 （BSCH.BVSCH.BVMAP,BVCPY.SCMP.SMOV.QSCH ）
r	/F=0./R=1 （SSCH）
c	/N=0./S=1 （CHK ） -- CHK. change index value の'c'
d	/0=0./1=1 （BSCH.BVSCH） -- dataの'd'
m	/NM=0./MR=1 （QSCH） -- maskの'm'
p	/AS=0./SS=1 （PTLB.PSTLB.LDATE） -- PTLB. sPeci fic space の'p'
ttt	/PT=000./ST=001./AT=110. /reserved=010 ～101,111(PSTLB.LDATE.STATE)
xx	/LS=00./CS=01 reserved=10.11(LDCTX.STCTX)

以上の項目に当てはまらないフィールド名は、オペランドフィールド名を示すものになる。できるだけ、同じ文字が複数の意味を表わさないようにしている。

【００９６】１１－４．オペランドフィールド名

オペランドフィールド名を表わす文字には、以下のような意味を持たせている。オペランドフィールド名はこれらの文字の組み合わせによって構成されるため、フィールド名だけで、使用できるアドレッシングモード、オペランドサイズ、アクセス方法などの情報を得ることができる。

47

・基本となるアドレッシングモード

Ea　８ビットの一般形アドレッシングモードを使用
Sh　６ビットの短縮形アドレッシングモードを使用
#　リテラル
#i　イミディエート
#d　ディスプレースメント
Rg　レジスタ
Ll　レジスタリスト（LDM 用）
Ls　レジスタリスト（STM 用）
Ln　レジスタリスト（ENTER 用）
Lx　レジスタリスト（EXITD 用）

48

＊【００９７】・アクセス方法

一部の基本アドレッシングモードでは、以下のようなアクセス方法がデフォルトとして決まっている。この場合には、特にアクセス方法を示す文字を付けない。

#.#i.#d　命令空間からのread
Ls.Ln　レジスタのread
Ll.Lx　レジスタへのwrite

その他の基本アドレッシングモードについては、以下に示す文字を使用してアクセス方法を示す。

R　read
W　write
M　read-modify-write

なお、フィールド名を短縮するため、RgR をRRに、RgW をRWに、RgM をRMに省略することがある。（BF命令、CSI 命令）

A　アドレス計算のみを行なう。
f　ビットオフセットとの組み合わせによって実際に操作を行なうメモリアドレスが決まる。（R .Mのサフィックス）
例：ビット操作命令
fq　ビットオフセットが付くが、ビットオフセットはバイト境界を越えない。アクセスすべきアドレスは、オフセットを見なくても確定している。（R.M のサフィックス）
例：短縮形のビット操作命令
bf　ビットオフセットやビットフィールド幅との組み合わせによって実際に操作を行なうメモリアドレスの範囲が決まる。（R.M のサフィックス）
例：固定長ビットフィールド操作命令
q　キュー命令による複雑なアクセスを行なう。（他のアクセス方法のサフィックス）
例：QINS.QDEL 命令
j　バスのインタロックによるアクセスを行なう。（M のサフィックス）
%　制御空間、物理空間などの特殊空間のアクセスを行なう。（R.W.M のサフィックス）
d　２つのデータ(double)に対する操作を行なう。（R のサフィックス）
例：CHK 命令
m　複数のデータ(multiple)に対する操作を行なう。（R.W のサフィックス）
例：LDM.STM 命令

・アドレッシングモードに対する制限

基本アドレッシングモードとアクセス方法が決まると、自動的にアドレッシングモードに対する制限（ＥａＷに対するイミディエートモードの禁止など）が決まる。ただし、それ以外に命令特有の制限事項がある場合には、以下の文字を後ろに付ける。

49

!I　イミディエートモードの禁止

例：CMP 命令の第二オペランド

!M　メモリ対象アドレッシングモードの禁止

例：ENTER:G 命令のlocal オペランド

!A　付加モードの禁止

例：LDCTX 命令のctxaddr オペランド

!S　スタックポップ、スタックプッシュモードの禁止

例：QDEL命令のdestオペランド

50

＊【００９８】・サイズ指定

サイズ指定は、原則として以下に示すフィールドによって行なう。

＊10

アクセス方法がR

RRフィールド

アクセス方法がW

WWフィールド

アクセス方法がM

MMフィールド

アクセス方法がR!I.R!M.R2

SSフィールド

アクセス方法が*f

BBフィールド

ただし、これはメモリ操作のアクセスサイズを意味する。

アクセス方法がA

サイズは指定されない。

【００９９】これより例外がある場合には、以下の文字を付け加えることにより区別する。原則として、数字と小文字が固定サイズを表わし、大文字は可変サイズを表わす。例えば、’ｗ’は３２ビット（ワード）固定のサイズを示すのに対して、’Ｗ’はWWフィールドによりサイズが指定されることを示す。

51　　52

W　オペランドサイズは必ず32ビット

例：MUL:R 命令

h　オペランドサイズは必ず16ビット

例：WAIT命令

b　オペランドサイズは必ず８ビット

例：MOV:E 命令のsrc

S8　オペランド（ディスプレースメント）のサイズは、SSフィールドにより指定される。ただし、このオペランド指定フィールドを使うのは、SS=00 （８bit 指定）の時に限られる。それ以外の場合には、拡張部によってオペランドが指定され、このフィールドは無視される。（０にしておくこと）

例：BF:I命令のsrc

S　オペランド（ディスプレースメント）のサイズは、SSフィールドにより指定される。

例：BRA:G 命令

R　オペランドサイズは、もう一方のオペランドのサイズと共通にRRフィールドにより指定される。

例：CMP:I 命令

W　オペランドサイズは、もう一方のオペランドのサイズと共通にWWフィールドにより指定される。

例：MOV:I 命令

M　オペランドサイズは、もう一方のオペランドのサイズと共通

53　　54

にMMフィールドにより指定される。

例：:Iフォーマットの命令

L　オペランドサイズとして８ビットまたは16ビットを指定するためのビットパターンが割り当てられていないため、32ビットまたは64ビットのオペランドのみが指定できる。
サイズ指定は、RR.WW,MM.BB フィールドではなく、R.M,W.B フィールドにより行なわれる。

P　ポインタを扱うため、命令中ではサイズ指定を行なわない。実際のサイズ指定は、P ビットまたはモード（PSW 中のXAビット）等によって行なわれる。

例：QINS.QDEL 命令

X　オペランドサイズは、XXフィールドにより指定される。

例：ACB.SCB 命令のxreg

Xw　オペランドサイズは、 Xフィールドにより他のオペランドと共通に指定される。これは、BF命令のwidth 指定用である。

Xs　オペランドサイズは、 Xフィールドにより他のオペランドと共通に指定される。これは、BF命令のsrc 指定用である。

Xd　オペランドサイズは、 Xフィールドにより他のオペランドと共通に指定される。これは、BF命令のdest指定用である。

C　オペランドサイズは、RRフィールドにより他のオペランドと共通に指定される。これは、CSI 命令の比較値指定用である。

3　3 ビットのリテラル

4　4 ビットのリテラル

例：TRAPA 命令

6　6 ビットのリテラル

8　8 ビットのディスプレースメント

例：BRA:8 命令

16　16ビットのディスプレースメント

例：MOVA:R命令

なお、ストリング命令などの高機能命令において、命令によって暗黙に指定されるオペランドのサイズを指定する場合には、フィールド名としてＳＳを使用する。任意長ビットフィールド命令では、Ｘも使用される。

【０１００】・その他

55 56

Z　リテラルで、ビットパターンの0 をオペランド値の0(zero)に対応させる場合を示す。ビットパターンとオペランド値との対応は、以下のようになる。（N はリテラルのビット数）

```
0...000     0
0...001     1
0...010     2
    ...
1...110     2^N-2
1...111     2^N-1
```

例：BTST:Qのoffset

n　リテラルで、ビットパターンの0 をオペランド値の 2^N に対応させる場合を示す。ビットパターンとオペランド値との対応は、以下のようになる。（N はリテラルのビット数）

```
0...000     2^N
0...001     1
0...010     2
    ...
1...110     2^N-2
1...111     2^N-1
```

例：MOV:Q のsrc

c　リテラルで、ビットパターンが２の補数(complement)を現わす場合を示す。ビットパターンとオペランド値との対応は、以下のようになる。（Nはリテラルのビット数）

```
0...000     -2^N
0...001     -( 2^N-1)
0...010     -( 2^N-2)
    ...
1...110     -2
1...111     -1
```

例：SHA:C.SHL:C で右シフトの場合のシフトカウント

1，2．．．一つの命令の中で、同じアクセス方法を持＊つオペランドが複数存在した場合に、それらを区別するために使用する。なお、サイズに関する種々の制限事項のうち、命令機能に大きな関係を持つものについては、オペランドフィールド名やサイズ指定フィールド名ではなく、各命令の説明のところでその制限を示す。これには、シフトカウントで８ビット以外のサイズを指定した場合や、異種サイズ間の論理演算などが含まれる。

【０１０１】１１－５．アドレッシングモードに関する制限

オペランドフィールド名のうち、次のものは、使用できるアドレッシングモードに制限が設けられている。

EaR.ShR

@-SPは利用できない。

EaW.ShW

#imm__data.@SP+ は利用できない。

EaM.ShM

#imm__data,@-SP,@SP+は利用できない。

EaA

@SP+,@-SP,Rn, #imm__dataは利用できない。

このほか、各命令の説明のところでもアドレッシングモードに関する制限が述べられている。

【０１０２】１１－６．解説に関する注意

スタック操作の命令では、ＴＯＳによりスタックトップを示している。↑ＴＯＳはスタックからのポップ、↓ＴＯＳはスタックへのプッシュである。

57 58

２オペランドの基本命令(MOV, MOVU, ADD, ADDU, ADDX, SUB, SUBU, SUBX, AND, OR, XOR, CMP, CMPU)では、次のような記法でそのオペレーションを説明している。

dest(src2)のサイズ（ビット数）をd で、src(src1) のサイズ（ビット数）をs で表わし、src(src1), dest(src2)をビット分解した値をD0, D1, ... Dd-1, S0, S1, ... Ss-1 で表わす。したがって、

dest(src2) = [D0. D1.... Dd-2. Dd-1]

src(src1) = [S0. S1.... Ss-2. Ss-1]

と書ける。[..]は２進数による表示を、'.' は各桁の区切りを意味する。この時、演算結果によってdestに設定される値を

dest .op. src = result

= [R0. R1.... Rd-2. Rd-1]

で表わす。MOV, MOVU, CMP, CMPU 以外の命令では、resultがdestにセットされる。またs >d の場合は、一般に演算結果の下位ビットのみをdestに設定することになる。この時、演算結果の上位ビットをカットする前の値を

result = [F0. F1.... Fs-2. Fs-1]

で表わす。R のビット数はｄ、F のビット数はs である。

ビット列[..]を符号付きの２進数と解釈した場合には、そのビット列の表現する値をS[..] で表わし、符号なしの２進数と解釈した場合には、そのビット列の表現する値をU[..] で表わす。また、そのビット列を符号付きPACKED形式10進数と解釈した場合には、そのビット列の表現する値をSD[..]で表わし、符号なしPACKED形式10進数と解釈した場合には、そのビット列の表現する値をUD[..]で表わす。さらに、

【０１０３】

【数５】

' - ' は真偽反転を、' ^ ' は累乗を表わす。

【０１０４】

同じように、固定長ビットフィールド命令の説明では、

bitfield = [Bo. Bo+1.... Bo+w-2. Bo+w-1]

といった記法でビット単位の詳細なオペレーションを説明している。

略記法として、

[Sn. Sn+1.... Sm-2. Sm-1] を

[Sn ～m-1]

で表わし、

[S0. S1..... Sd-2. Sd-1]

59　　60

= [S0 ～s-1]

を単に

[S]

で表わすことがある。[D]，[R]，[B]，[F] についても同様である。

【０１０５】１２．「本発明装置」の命令セット

【０１０６】１２－１．データ転送命令

〔ニモニック〕

ＭＯＶ　　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｓｒｃ＝＝〉ｄｅｓｔ

データの移動と符号拡張

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図４８に示す。

〔フラッグ変化〕図４８の最下部に示す。

〔解説〕ソースオペランドｓｒｃをデスティネーションオペランドｄｅｓｔに転送する。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースが符号拡張される。デスティネーションの方がサイズが小さく、ソースの値がデスティネーションのサイズの符号付き整数として表現できない時は、Ｖ＿ｆｌａｇがセットされる。ＭＯＶ：Ｚはいわゆるｃｌｅａｒ命令であるが、動作やフラッグ変化が同じであるため、ＭＯＶの短縮形の一つとして扱っている。ＭＯＶ，ＡＤＤ，ＭＯＶ，ＣＭＰ命令は符号付きの演算を行なう命令であるが、ＭＯＶ：Ｑ，ＡＤＤ：Ｑ，ＳＵＢ：Ｑ，ＣＭＰ：Ｑで利用できるリテラルの範囲は１～８（オペランドフィールド名＃３ｎ）となっており、正の範囲しか含んでいないので、注意が必要である。ＭＯＶ，ＭＯＶＵ命令でｓｒｃがイミディエート値である場合に、そのイミディエート値と利用できるフォーマットとの関係をまとめると、次のようになる。

[MOV]

:Z	src ＝0
:Q	1≦src ≦8
:B	-128≦src ≦127
:I	src は任意
:G	src は任意

[MOVU]

:B	0≦src ≦255
:G	src は任意

ADD，SUB，CMP 命令も同様である。

61 62

(d ≧s の時)

[S0. S1....Ss-2.Ss-1] ==>

[S0.S0.....S0.

S0. S1....Ss-2.Ss-1] ==>

d-s ビットだけ符号拡張

[R0.R1.....Rd-s+1.Rd-s.Rd-s+1....Rd-2.Rd-1] (destに設定される)

(d <s の時)

[S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

[Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

S0.S1.....Ss-d-1

s-d ビットがカットされる。

[R0. R1. Rd-2.Rd-1] (destに設定される)

M__flag R0

Z__flag [R0 ～d-1] = 0

V__flag☆ S[S] < -2^(d-1) .or. S[S] ≧ +2^(d-1)

つまり、d ≧s であればクリアされ、d <s であれば、

S0 = S1 =.....= Ss-d-1 = Ss-d(=R0)

の時クリア、それ以外の場合にセットとなる。

【０１０７】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・WW='11' のとき

・BaR,ShR が@-SPのとき

・BaW,ShW が#imm__data,@SP+ のとき

〔ニモニック〕

MOVU src,dest

〔命令の機能〕

zex(src) ==> dest

データの移動とゼロ拡張

(d ≧s の時)

[S0. S1....Ss-2.Ss-1] ==>

[0. 0.....0. S0. S1....Ss-2.Ss-1] ==>

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図４９に示す。

〔フラッグ変化〕図５０に示す。

〔解説〕ソースオペランドｓｒｃをデスティネーションオペランドｄｅｓｔに転送する。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースがゼロ拡張される。デスティネーションの方がサイズが小さく、ソースの値がデスティネーションのサイズの符号なし整数として表現できない時は、Ｖ__ｆｌａｇがセットされる。

d-s ビットだけゼロ拡張

[R0.R1.....Rd-s+1.Rd-s.Rd-s+1....Rd-2.Rd-1] （destに設定される）

（d ＜s の時）

[S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

_____[Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

S0.S1.....Ss-d-1

s-d ビットがカットされる。

	[R0. R1. Rd-2.Rd-1 （destに設定される）
M__flag	R0
Z__flag	[R0～d-1] = 0
V__flag☆	U[S] ≧ +2 ^d

つまり、d ≧s であればクリアされ、d ＜s であれば、

S0 = S1 =.....= Ss-d-1 = 0

の時クリア、それ以外の場合にセットとなる。

【０１０８】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・WW='11' のとき

・EaR が@-SPのとき

・EaW が#imm__data,@SP+ のとき

〔ニモニック〕

ＰＵＳＨ　ｓｒｃ

〔命令の機能〕

ｐｕｓｈ　ｔｏ　ｓｔａｃｋ

スタックにプッシュ

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図５１に示す。

〔フラッグ変化〕図５２に示す。

〔解説〕ソースオペランドｓｒｃをスタックにプッシュする。この命令は、ＭＯＶ　＊，@－ＳＰの短縮形と考えることもできるが、フラッグ変化をしないこと、およびＰＯＰとの対称性により、別命令となっている。ｓｒｃ／ＥａＲＬで指定されるアドレッシングモードでは、@ＳＰ＋のモードは使用できない。これは、ＰＯＰ命令のｄｅｓｔ／ＥａＷＬで@－ＳＰのモードが使用できないのに合わせたものである。ＰＵＳＨ　ＳＰなど、ｓｒｃオペランドにＳＰを含む場合の命令動作規定については、付録１２を参照のこと。

【０１０９】〔プログラム例外〕

・予約命令例外

・R='1' のとき

・EaRLが@SP+,@-SP のとき

〔ニモニック〕

ＰＯＰ　ｄｅｓｔ

〔命令の機能〕

ｐｏｐ　ｆｒｏｍ　ｓｔａｃｋ

スタックからポップ

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図５３に示す。

〔フラッグ変化〕図５４に示す。

〔解説〕スタックからポップした値をｄｅｓｔに転送する。この命令は、ＭＯＶ　@ＳＰ＋，＊の短縮形と考えることもできるが、ｓｒｃにＳＰを含んだ場合の動作がＭＯＶ　@ＳＰ＋とは異なること、およびフラッグ変化をしないこと、によって別命令となっている。ｄｅｓｔ／ＥａＷＬで指定されるアドレッシングモードでは、@－ＳＰのモードを使用することは禁止されており、指定した場合には予約命令例外ＲＩＥとなる。これは、ＰＯＰ　@－ＳＰという命令を実行した場合に、ＳＰ更新がいつ行なわれるかという点について誤解を生じやすいためである。ＰＯＰ　ＳＰなど、ｄｅｓｔオペランドにＳＰを含む場合の命令の動作規定については、付録１２を参照のこと。

【０１１０】〔プログラム例外〕

・予約命令例外

・W='1' のとき

・EaWLが#imm__data,@SP+,@-SPのとき

〔ニモニック〕

ＬＤＭ　ｓｒｃ，ｒｅｇｌｉｓｔ

〔命令の機能〕

ｌｏａｄ　ｍｕｌｔｉｐｌｅ　ｒｅｇｉｓｔｅｒｓ

複数レジスタのロード

〔命令オプション〕なし

65

〔命令ビットパターンとアセンブラ表記〕図５５に示す。

〔フラッグ変化〕図５６に示す。

〔解説〕複数のレジスタをメモリからロードする。ロードするレジスタはビットマップｒｅｇｌｉｓｔ／ＬＩＲＬ（レジスタリスト）で指定する。ＬＩＲＬは、ＥａＲｍＬの拡張部よりも後に置かれる。ロードするレジスタリスト（ｒｅｇｌｉｓｔ）のビットマップ指定は、図５７に示すように行なう。ＥａＲｍＬで@ＳＰ＋のアドレッシングモードを指定した場合は、小さい番号のレジスタから順にポップされ、ＳＰはロードしたレジスタ数の４倍（または８倍）だけ増加する。それ以外のアドレッシングモードを指定した場合は、得られた実効アドレスがレジスタにロードすべきメモリデータの先頭を指す。いずれの場合にも、メモリ中では小さい番号のレジスタの方が低いアドレスに置かれる。ロードするレジスタのビットマップのフォーマットは、ＢＳＣＨ／Ｆ，ＢＶＳＣＨ／Ｆ命令で使用する回路（次に出現する’０’または’１’のビットをＭＳＢ方向にサーチする回路）と同じ回路によって、次に転送するレジスタを見付けられるように決めたものである。したがって、ＬＤＭ　@ＳＰ＋の場合は小さな番号のレジスタから転送するためにレジスタ番号の小さな方がＭＳＢ側となっている。それ以外のアドレッシングモードの場合にも、レジスタ退避ブロックの先頭アドレスを実効アドレスとしているため、やはりレジスタ番号の小さい方から転送するのがよく、ＬＤＭ　@ＳＰ＋と同じフォーマットになる。なお、これらのフォーマットはレジスタの転送順序まで考えて決めたものであり、ハードウエア資源が少ない場合には、ここで説明したような転送順序にするのが最適と考えられる。しかし、実際の転送の順序は「本発明装置」で規定されたものではなく、インプリメント側の自由である。ＥａＲｍＬのアドレッシングモードでは、@－ＳＰ、レジスタ直接モードＲｎ，イミディエートモード#ｉｍｍ＿ｄａｔａ、付加モードの指定はイリーガルとする。付加モードを禁止するのは、ＬＤＭやＳＴＭによって退避、復帰したレジスタやレジスタ退避エリアと、付加モードで使用するレジスタやメモリの間にオーバーラップがあった場合に、命令の再実行が難しくなるためである。レジスタリストがオール０の時は、何もせずに命令を終了する。（特にエラーとはしない）

ＬＤＭ　@ＳＰ＋でレジスタリストにＳＰが含まれる場合の動作規定については、付録１２を参照のこと。

【０１１１】〔プログラム例外〕

・予約命令例外

・R='1' のとき

・EaRmL が Rn, #imm_data, @-SP, 付加モードのとき

〔ニモニック〕

ＳＴＭ　ｒｅｇｌｉｓｔ，ｄｅｓｔ

66

〔命令の機能〕

ｓｔｏｒｅ　ｍｕｌｔｉｐｌｅ　ｒｅｇｉｓｔｅｒｓ

複数レジスタのストア

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図５８に示す。

〔フラッグ変化〕図５９に示す。

〔解説〕複数のレジスタをメモリにセーブする。セーブするレジスタはビットマップｒｅｇｌｉｓｔ／ＬｓＷＬ（レジスタリスト）で指定する。ＬｓＷＬは、ＥａＷｍＬの拡張部よりも後に置かれる。ストアするレジスタリスト（ｒｅｇｌｉｓｔ）のビットマップ指定は、ＥａＷｍＬが@－ＳＰモードの時、図６０に示すように、またその他のモードの時、図６１に示すように行う。ＥａＷｍＬに@－ＳＰのアドレッシングモードを指定した場合は、大きい番号のレジスタから順にプッシュされ、ＳＰはセーブしたレジスタ数の４倍（または８倍）だけ減少する。それ以外のアドレッシングモードを指定した場合は、得られた実効アドレスがレジスタをセーブすべきメモリ領域の先頭を指す。いずれの場合にも、メモリ中では小さい番号のレジスタの方が低いアドレスに置かれる。このフォーマットは、ＢＳＣＨ／Ｆ，ＢＶＳＣＨ／Ｆ命令で使用する回路（次に出現する’０’または’１’のビットをＭＳＢ方向にサーチする回路）と同じ回路によって、次に転送するレジスタを見付けられるように決めたものである。したがって、ＳＴＭ　@－ＳＰの時は大きな番号のレジスタから転送するためにレジスタ番号の大きな方がＭＳＢ側となる。それ以外のアドレッシングモードの場合には、レジスタ退避ブロックの先頭アドレスを実効アドレスとしているため、レジスタ番号の小さい方から転送するのがよく、レジスタ番号の小さな方がＭＳＢ側となっている。なお、これらのフォーマットはレジスタの転送順序まで考えて決めたものであり、ハードウエア資源が少ない場合には、ここで説明したような転送順序にするのが最適と考えられる。しかし、実際の転送の順序は「本発明装置」で規定されたものではなく、インプリメント側の自由である。ＥａＷｍＬのアドレッシングモードでは、@ＳＰ＋、レジスタ直接モードＲｎ、イミディエートモード#ｉｍｍ＿ｄａｔａ、付加モードの指定はイリーガルとする。付加モードを禁止するのは、ＬＤＭやＳＴＭによって退避、復帰したレジスタやレジスタ退避エリアと、付加モードで使用するレジスタやメモリの間にオーバーラップがあった場合に、命令の再実行が難しくなるためである。ＬＤＭ，ＳＴＭ命令では、転送しないレジスタに対するメモリ領域は割り当てない。例えば、

STM.W　(R1,R3,R9),@-SP

の場合は次のような動作を行なう。（ただし、命令実行前のＳＰ値をｉｎｉｔＳＰとする。）

67

R9 ==> mem[initSP - 4]

R3 ==> mem[initSP - 8]

R1 ==> mem[initSP - 12]

initSP - 12 ==> SP

レジスタリストがオール０の時は、何もせずに命令を終了する。（特にエラーとはしない）
ＳＴＭ　@－ＳＰでレジスタリストにＳＰが含まれる場合の動作規定については、付録１２を参照のこと。
【０１１２】〔プログラム例外〕
・予約命令例外
・W='1' のとき
・EaWmL がRn, #imm__data,@SP+,付加モードのとき
〔ニモニック〕
ＭＯＶＡ　ｓｒｃａｄｄｒ，ｄｅｓｔ
〔命令の機能〕
ａｄｄｒｅｓｓ　ｏｆ　ｓｒｃ＝＝）ｄｅｓｔ
実効アドレスを得る
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図６２に示す。
〔フラッグ変化〕図６３に示す。
〔解説〕ソースオペランドの実効アドレスをデスティネーションオペランドに転送する。この命令のオペレーションそのものは、ＭＯＶ命令などで代用可能であるが、高級言語での左辺値のアドレス計算やポインタ演算にすなおに使用できること、アドレス計算用の回路を使用するため、より高速な演算が期待できること、により別命令となっている。短縮形の

MOVA:R　@(disp:16,Rs),Rd

は、実質的には３オペランド加算命令

Rs + disp:16 -> Rd

となるが、フラッグ変化をおこさないためＭＯＶＡ命令に分類されている。ｓｒｃａｄｄｒにＰＣ相対間接モードを指定し、ＰＣ相対のディスプレースメントを０とした場合には、現在のＰＣ値、つまりＭＯＶＡ命令の先頭アドレスをｄｅｓｔに格納することになる。また、ＰＣ相対のディスプレースメントとしてＭＯＶＡ命令の命令長を指定した場合には、ＭＯＶＡ命令の次の命令のアドレスをｄｅｓｔに格納することになる。これらの機能は、ユーザプログラムのレベルでコルーチン処理を行なう時に有効である。アセンブラでは、＜オペレーション＞またはｄｅｓｔ側でサイズ指定を行なう。ｓｒｃａｄｄｒ側はアドレス計算のみなので、サイズの指定はしない。ＥａＡで指定されるアドレッシングモードでは、イミディエート、@ＳＰ＋，@－ＳＰのモードは使用できない。
【０１１３】〔プログラム例外〕

68

・予約命令例外
・+='0' のとき
・W='1' のとき
・EaA がRn, #imm__data,@SP+,@-SPのとき
・EaW が#imm__data,@SP+ のとき
〔ニモニック〕
ＰＵＳＨＡ　ｓｒｃａｄｄｒ
〔命令の機能〕
ｐｕｓｈ　ａｄｄｒｅｓｓ　ｔｏ　ｓｔａｃｋ
実効アドレスをスタックにプッシュ
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ記法〕図６４に示す。
〔フラッグ変化〕図６５に示す。
〔解説〕ソースオペランドｓｒｃａｄｄｒの実効アドレスをスタックにプッシュする。この命令は、ＭＯＶＡ＊，@－ＳＰの短縮形と考えることもできるが、実行速度の向上や、ＭＯＶ命令～ＰＵＳＨ命令の区別との対応をとるために、別命令となっている。ｓｒｃにＳＰを含んだ場合の動作規定については、付録１２を参照のこと。
【０１１４】〔プログラム例外〕
・予約命令例外
・S='1' のとき
・EaA がRn, #imm__data,@SP+,@-SPのとき

【０１１５】１２－２．比較、テスト命令
〔ニモニック〕
ＣＭＰ　ｓｒｃ１，ｓｒｃ２
〔命令の機能〕
ｓｒｃ２－ｓｒｃ１，ｆｌａｇｓ　ａｆｆｅｃｔｅｄ
比較、符号拡張と比較
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図６６に示す。
〔フラッグ変化〕図６７に示す。
〔解説〕ｓｒｃ１オペランドをｓｒｃ２オペランドと比較し、その結果によりＰＳＢ（Ｌ＿ｆｌａｇ，Ｚ＿ｆｌａｇ）をセットする。ｓｒｃ１オペランドのサイズとｓｒｃ２オペランドのサイズが異なる時は、サイズの小さい方のオペランドが符号拡張された上で比較される。なお、ＥａＲ！Ｉ，ＳｈＲ！Ｉのモードではイミディエートを禁止しているが、@ＳＰ＋は可能である。ＣＭＰ@ＳＰ＋，@ＳＰ＋の場合、スタックポインタはオペランドサイズの２倍だけ変化し、他の命令と比較すると変則的であるが、この命令はスタックマシンをシミュレートする場合に使用することがある。
ＣＭＰ：Ｚはいわゆるｔｅｓｔ命令であるが、動作やフラッグ変化が同じであるため、ＣＭＰの短縮形の一つと

69 70

して扱っている。

以下では、

src1 = [S0.S1.....Ss-2.Ss-1]

src2 = [D0.D1.....Dd-2.Dd-1]

によってオペレーションを説明する。

（d ≧s の時）

[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] -

[S0.S0.....S0. S0. S1....Ss-2.Ss-1] ==>

d-s ビットだけ符号拡張

[R0.R1.....Rd-s-1.Rd-s.Rd-s+1....Rd-2.Rd-1]（どこにも設定されない）

（d <s の時）

[D0.D0.....D0. D0. D1....Dd-2.Dd-1] -

s-d ビットだけ符号拡張

[S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

[F0.F1.....Fs-d-1.Fs-d.Fs-d+1....Fs-2.Fs-1]（どこにも設定されない）

L__flag☆ S[D] < S[S]
SUB命令と同じ

Z__flag [R0～d-1] = 0
（d ≧s の時）

☆ [F0～s-1] = 0
（d <s の時）

【０１１６】〔プログラム例外〕
・予約命令例外
・RR='11' のとき

・SS='11' のとき
・EaR.ShR が@-SPのとき
・EaR!I.ShR!I が#imm__data,@-SP のとき

〔ニモニック〕
CMPU ｓｒｃ１，ｓｒｃ２
〔命令の機能〕
ｓｒｃ２－ｓｒｃ１，ｆｌａｇｓ ａｆｆｅｃｔｅｄ

ゼロ拡張と比較
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図６８に示す。
〔フラッグ変化〕図６９に示す。
〔解説〕ｓｒｃ１オペランドをｓｒｃ２オペランドと比較し、その結果によりＰＳＢ（Ｌ__ｆｌａｇ，Ｚ__ｆｌａｇ）をセットする。ｓｒｃ１オペランドのサイズとｓｒｃ２オペランドのサイズが異なる時は、サイズの小さい方のオペランドがゼロ拡張された上で比較される。ＥａＲ！Ｉのモードではイミディエートを禁止しているが、@ＳＰ＋は可能である。

71　　72

以下では、

src1 = [S0.S1.....Ss-2.Ss-1]

src2 = [D0.D1.....Dd-2.Dd-1]

によってオペレーションを説明する。

（d ≧s の時）

[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] -

[0. 0.....0 S0. S1....Ss-2.Ss-1] ==>

d-s ビットだけゼロ拡張

[R0.R1.....Rd-s-1.Rd-s.Rd-s+1....Rd-2.Rd-1] （どこにも設定されない）

（d <s の時）

[0. 0.....0. D0. D1....Dd-2.Dd-1] -

s-d ビットだけゼロ拡張

[S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

[F0.F1.....Fs-d-1.Fs-d.Fs-d+1....Fs-2.Fs-1] （どこにも設定されない）

L__flag☆　U[D] < U[S]
　　SUBU 命令と同じ
Z__flag　[R0～d-1] = 0
　　(d ≧s の時)
　☆　[F0～s-1] = 0
　　(d <s の時)

【０１１７】〔プログラム例外〕
・予約命令例外
　・RR='11' のとき
　・SS='11' のとき
　・EaR が@-SPのとき
　・EaR[[が#imm__data.@-SP のとき

〔ニモニック〕
ＣＨＫ　ｂｏｕｎｄ，ｉｎｄｅｘ，ｘｒｅｇ
〔命令の機能〕
ｃｈｅｃｋ　ｕｐｐｅｒ　ａｎｄ　ｌｏｗｅｒ　ｂｏｕｎｄｓ
配列の範囲のチェック
〔命令オプション〕
／Ｓ　　下限値を引く
／Ｎ　　下限値を引かない（デフォルト）
〔命令ビットパターンとアセンブラ表記〕図７０に示す。

〔フラッグ変化〕図７１に示す。
〔解説〕配列のインデクスの範囲のチェックとレジスタへのロードを行なう。ｂｏｕｎｄの指すアドレスには上限値と下限値が組みになって置かれており、その上限値、下限値とｉｎｄｅｘによりフェッチされた比較値オペランドが比較される。ｂｏｕｎｄの実効アドレスに置かれているのが上限値であり、（ｂｏｕｎｄの実効アドレス＋オペランドサイズ）のアドレスに置かれているのが下限値である。比較は符号付き整数として行なわれる。比較値が上限値と下限値の間に入っていない場合には、Ｖ__ｆｌａｇがセットされるので、続けてＴＲＡＰ命令を実行することにより、例外処理を起動することができる。／Ｓを指定した場合、比較値から下限値を引いたものがレジスタｘｒｅｇにロードされる。／Ｓを指定しない場合、比較値はそのままレジスタｘｒｅｇにロードされる。比較値をレジスタにロードするのは、次にそれを配列のインデクスのアドレス計算に使うことが多いためである。
オペレーション：

```
        tmp = mem[address__of__bound +  operand__size]
    if (index ≧ mem[address__of__bound] .or. index < tmp)
       then
          set V__flag:
       if (A == 1)
       then
          index - tmp ==> xreg
       else
          index ==> xreg
```

ただし、'address__of__' は'mem[..]' の逆演算子であり、bound とmem [address__of__bound]が同じ意味になる。

下限の方は、値が一致した場合に範囲内と見なされるが、上限の方は、値が一致した場合には範囲外と見なされる。例えばbound のメモリが(0,100) となっていると、CHK で範囲内となるのはindex が０～99の場合である。

L__flag, Z__flagは、下限値とindex との比較結果にしたがってCMPと同様にセットされるが、L__flag=1となるのは、

ｉｎｄｅｘ＜下限値

の場合である。つまり、図７２のようになる。

①上限値＜下限値の場合には、下限値との比較により１になることもある。この場合、ｉｎｄｅｘ－下限値の演算結果によってフラッグがセットされることになる。次の３つの命令は、すべて第二オペランドが第一オペランド（ＣＨＫでは第一オペランドｂｏｕｎｄの下限値）よりも小さい時にＬ＿ｆｌａｇがセットされるという仕様である。

```
CMP    src1.src2
SUB    src.dest
CHK    bound.index.xreg
```

ＣＨＫ命令では、上限値≧下限値のチェックは特に行なわない。上限値と下限値の大小にかかわらず、「オペレーション」に書かれたものと等価の動作を行なうものとする。ＥａＲｄＲで指定されるアドレッシングモードでは、レジスタ直接Ｒｎ，＠－ＳＰ，＠ＳＰ＋，＃ｉｍｍ＿ｄａｔａのモードは使用できない。どうしてもレジスタ上の値と比較したい場合には、ＣＨＫではなくＣＭＰを２回行なえばよい。

【０１１８】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・EaR が@-SPのとき

・EaRdR がRn, #imm__data,@SP+,@-SPのとき

【０１１９】１２－３．算術演算命令

〔ニモニック〕

ＡＤＤ　　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｄｅｓｔ＋ｓｒｃ＝＝〉ｄｅｓｔ

加算、符号拡張と加算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図７３に示す。

〔フラッグ変化〕図７４に示す。

〔解説〕ソースオペランドをデスティネーションオペランドに加算する。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースが符号拡張された上で加算される。デスティネーションのサイズが小さく、演算結果がデスティネーションのサイズの符号付き整数として表現できない時は、Ｖ＿ｆｌａｇがセットされる。なお、Ｌ＿ｆｏｒｍａｔのＡＤＤ：Ｌ　＠ＳＰ＋，ＳＰについては、ＡＤＤ：Ｇ

(initSP+4)+@initSP==>SP

の動作を行なうのが望ましい。しかし、インプリメント上、Ｌ－ｆｏｒｍａｔではこのような動作を行なうのが難しい場合があるので、ＡＤＤ：Ｌ　＠ＳＰ＋，ＳＰの動作についてはインプリメント依存とする。これは、ＳＵＢ：Ｌ，ＣＭＰ：Ｌ，ＩＮＤＥＸも同様である。詳しくは付録１２を参照のこと。

75 76

（d ≧s の時）

[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] +

[S0.S0.....S0. S0. S1....Ss-2.Ss-1] ==>

d-s ビットだけ符号拡張

[R0.R1.....Rd-s-1.Rd-s.Rd-s+1....Rd-2.Rd-1] （destに設定される）

（d <s の時）

[D0.D0.....D0. D0. D1....Dd-2.Dd-1] +

s-dビットだけ符号拡張

[S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

[F0.F1.....Fs-d-1.Fs-d.Fs-d+1....Fs-2.Fs-1] ==>

[R0. R1....Rd-2.Rd-1] (destに設定される)

F0.F1.....Fs-d-1

s-dビットがカットされる

L__flag☆ S[D] + S[S]< 0

結果が負になることを表わす。

（ M__flagも結果の正負を表わすが、 M__flagが正しい正負を表示するのはオーバーフローのない時に限られる。）

M__flag R0

Z__flag [R0～d-1] = 0

V__flag S[D] + S[S]<-2^(d-1) .or. S[D] + S[S]≧ +2^(d-1)

X__flag☆ いずれの場合も、destのサイズからの桁上げが X__flagにセットされる。

（d ≧s の時）

U[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] +

U[S0.S0.....S0. S0.

S1....Ss-2.Ss-1]≧ +2^d

d-s ビットだけ符号拡張

（d <s の時）

U[D0. D1....Dd-2.Dd-1] +

U[Ss-d.Ss-d+1 ....Ss-2.Ss-1]

≧ +2^d

S0.S1.....Ss-d-1

s-d ビットがカットされる

【０１２０】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR.ShRwが@-SPのとき

・EaM.ShM が#imm__data.@SP+.@-SPのとき

〔ニモニック〕

ＡＤＤＵ　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｄｅｓｔ＋ｓｒｃ＝＝〉ｄｅｓｔ

ゼロ拡張と加算

〔命令オプション〕なし

50 〔命令ビットパターンとアセンブラ表記〕図７５に示

77

す。

〔フラッグ変化〕図７６に示す。

〔解説〕ソースオペランドをデスティネーションオペランドに加算する。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースがゼロ拡張された上で加算される。デスティネーショ＊

78

＊ンのサイズが小さく、演算結果がデスティネーションのサイズの符号なし整数として表現できない時は、Ｖ＿ｆｌａｇがセットされる。ＡＤＤＵのＬ＿ｆｌａｇは、結果が正になるという意味で、必ず０にリセットされるものとする。

（d ≧s の時）

[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] +

[0. 0.....0. S0. S1....Ss-2.Ss-1] ==>

d-s ビットだけゼロ拡張

[R0.R1.....Rd-s-1.Rd-s.Rd-s+1....Rd-2.Rd-1] （destに設定される）

（d ＜s の時）

[0. 0.....0. D0. D1....Dd-2.Dd-1] +

s-d ビットだけゼロ拡張

[S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

[F0.F1.....Fs-d-1.Fs-d.Fs-d+1....Fs-2.Fs-1] ==>

[R0. R1....Rd-2.Rd-1] (destに設定される)

F0.F1.....Fs-d-1

s-d ビットがカットされる。

L__flag	0
M__flag	R0
Z__flag	[R0 ～d-1] = 0
V__flag	U[D] + U[S] ≧ +2^d
X__flag☆	いずれの場合も、destのサイズからの桁上げが X__flag にセットされる。

（d ≧s の時）

U[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] +

U[0. 0.....0. S0.

S1....Ss-2.Ss-1]≧ +2^d

d-s ビットだけゼロ拡張

ADDU命令の V__flagと同じ

（d ＜s の時）

[D0. D1....Dd-2.Dd-1] +

U[Ss-d.Ss-d+1 ....Ss-2.Ss-1]

≧ +2^d

S0.S1.....Ss-d-1

s-dビットがカットされる

【０１２１】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

79

・EaR が@-SPのとき

・EaM が#imm__data,@SP+,@-SPのとき

〔ニモニック〕

ＡＤＤＸ　　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

dest + src + X__flag --> dest

キャリーを含めた加算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図７７に示す。

〔フラッグ変化〕図７８に示す。

80

〔解説〕ソースオペランドとキャリーをデスティネーションオペランドに加算する。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースが符号拡張された上で加算される。Ｚ__ｆｌａｇでは、フラッグ値を累積できるようになっている。また、ＡＤＤＸとＡＤＤのフラッグ変化は、符号拡張／ゼロ拡張を含めてほとんど同じである。ＡＤＤとＡＤＤＸでフラッグ変化の異なるのは、Ｚ__ｆｌａｇのみである。なお、ＡＤＤＸ，ＳＵＢＸの累種サイズ間演算については、例えば８バイトの数ｄｅｓｔ２～ｄｅｓｔ１に４バイトの数ｓｒｃを加える場合、

ADD　@src.W,@dest1.W

ADDX　#0,@dest2.W

のADDX:E #0 といった形で利用することがある。

(d ≧s の時)

[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] +

[S0. S0.....S0. S0. S1....Ss-2.Ss-1] + X__flag ==>

d-s ビットだけ符号拡張

[R0.R1.....Rd-s-1.Rd-s.Rd-s+1....Rd-2.Rd-1] (destに設定される)

(d <s の時)

[D0.D0.....D0. D0. D1....Dd-2.Dd-1] +

s-d ビットだけ符号拡張

[S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] + X__flag ==>

[F0.F1.....Fs-d-1.Fs-d.Fs-d+1....Fs-2.Fs-1] ==>

_____[R0. R1....Rd-2.Rd-1] (destに設定される)

F0.F1.....Fs-d-1

s-d ビットがカットされる。

L__flag☆　S[D] + S[S] + X__flag< 0

符号付きの数と見て演算を行ない、結果が負になることを表わす。d ≠s の場合には、オペランドが符号拡張されてから比較される。（M__flagも結果の正負を表わすが、M__flagが正しい正負を表示するのはオーバーフローのない時に限られる。）

M__flag　R0

Z__flag　[R0～d-1] = 0 .and. previous Z__flag

V__flag　S[D] + S[S] + X__flag< -2^(d-1) .or.

81　　82

S[D] + S[S] + X__flag≧ +2^(d-1)

符号付きの数と見て演算を行ない、結果がオーバーフローすることを表わす。d ≠s の場合にはオペランドが符号拡張される。

X__flag☆　いずれの場合も、destのサイズからの桁上げが X__flagにセットされる。

(d ≧s の時)

U[D0.D1.....Dd-s-1.Dd-s.Dd-s+1.... Dd-2.Dd-1] +

U[S0.S0.....S0. S0. S1.... Ss-2.Ss-1] + X__flag

≧ +2^d

d-s ビットだけ符号拡張

d>s の場合には符号拡張を行なう。これは、destや他のフラッグの設定の処理と共通化したためである。しかし、符号拡張後の演算、比較ではオペランドが符号なしの数として扱われる。

(d <s の時)

U[D0. D1....Dd-2.Dd-1] +

U[Ss-d.Ss-d+1 ....Ss-2.Ss-1] + X__flag

≧ +2^d

S0.S1.....Ss-d-1

s-d ビットがカットされる

【０１２２】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・BaR が@-SPのとき

・BaM が#imm__data.@SP+,@-SPのとき

〔ニモニック〕

ＳＵＢ　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｄｅｓｔ－ｓｒｃ＝＝〉ｄｅｓｔ

減算、符号拡張と減算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図７９に示す。

〔フラッグ変化〕図８０に示す。

〔解説〕ソースオペランドをデスティネーションオペランドから減ずる。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースが符号拡張された上で減算される。デスティネーションのサイズが小さく、演算結果がデスティネーションのサイズの符号付き整数として表現できない時は、Ｖ__ｆｌａｇがセットされる。

(d ≧s の時)

[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] -

[S0.S0.....S0. S0. S1....Ss-2.Ss-1] ==>

83 84

d-sビットだけ符号拡張

[R0.R1.....Rd-s-1.Rd-s.Rd-s+1....Rd-2.Rd-1]（destに設定される）

（d ＜s の時）

{D0.D0.....D0. D0. D1....Dd-2.Dd-1] -

s-dビットだけ符号拡張

{S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

{F0.F1.....Fs-d-1.Fs-d.Fs-d+1....Fs-2.Fs-1] ==>

[R0. R1....Rd-2.Rd-1]（destに設定される）

F0.F1.....Fs-d-1

s-dビットがカットされる。

L__flag☆ S[D] - S[S] ＜ 0

結果が負になることを表わす。

（ M__flagも結果の正負を表わすが、 M__flagが正しい正負を表示するのはオーバーフローのない時に限られる。）

M__flag R0

Z__flag [R0 ～d-1] = 0

V__flag S[D] - S[S] ＜ -2^(d-1) .or.

S[D] - S[S] ≧ +2^(d-1)

X__flag☆ いずれの場合も、destのサイズからの桁下げが X__flag にセットされる。

（d ≧s の時）

U[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] -

U[S0.S0.....S0. S0. S1....Ss-2.Ss-1]＜ 0

d-sビットだけ符号拡張

（d ＜s の時）

U[D0. D1....Dd-2.Dd-1] -

U[Ss-d.Ss-d+1 ....Ss-2.Ss-1]＜ 0

S0.S1.....Ss-d-1

s-dビットがカットされる

【０１２３】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR.ShRwが@-SPのとき

・EaM.ShM が#imm__data.@SP+.@-SPのとき

〔ニモニック〕

ＳＵＢＵ ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｄｅｓｔ－ｓｒｃ＝＝）ｄｅｓｔ

ゼロ拡張と減算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図８１に示す。

〔フラッグ変化〕図８２に示す。

〔解説〕ソースオペランドをデスティネーションオペランドから減ずる。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースがゼロ拡張された上で減算される。デスティネーションのサイズが小さく、演算結果がデスティネーションのサイズの符号なし整数として表現できない時は、Ｖ__ｆｌａｇがセットされる。

85 86

(d ≧s の時)

[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] -

[0. 0.....0. S0. S1....Ss-2.Ss-1] ==>

d-s ビットだけゼロ拡張

[R0.R1.....Rd-s-1.Rd-s.Rd-s+1....Rd-2.Rd-1] (destに設定される)

(d <s の時)

[0. 0.....0. D0. D1....Dd-2.Dd-1] -

s-d ビットだけゼロ拡張

[S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] ==>

[F0.F1.....Fs-d-1.Fs-d.Fs-d+1....Fs-2.Fs-1] ==>

_____[R0. R1....Rd-2.Rd-1] (destに設定される)

F0.F1.....Fs-d-1

s-dビットがカットされる。

L__flag☆ U[D] - U[S] < 0

結果が負になることを表わす。

(M__flagも結果の正負を表わすが、 M__flagが正しい正負を表示するのはオーバーフローのない時に限られる。)

M__flag R0

Z__flag [R0 ～d-1] = 0

V__flag U[D] - U[S] < 0

SUBU命令の L__flagと同じ

X__flag☆ いずれの場合も、destのサイズからの桁下げが X__flag にセットされる。

(d ≧s の時)

U[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] -

U[0. 0.....0. S0. S1...Ss-2.Ss-1] < 0

d-sビットだけゼロ拡張

SUB命令の X__flag、SUBU命令の L__flag,V__flagと同じ

(d <s の時)

U[D0. D1....Dd-2.Dd-1] -

_____U[Ss-d.Ss-d+1 ....Ss-2.Ss-1]< 0

S0.S1.....Ss-d-1

s-dビットがカットされる

【0124】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm__data.@SP+,@-SPのとき

50

87

〔ニモニック〕

ＳＵＢＸ　　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

dest - src - X__flag ==> dest

キャリーを含めた減算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図８３に示す。

〔フラッグ変化〕図８４に示す。

88

〔解説〕ソースオペランドとキャリーをデスティネーションオペランドから減ずる。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースが符号拡張された上で減算される。Ｚ＿ｆｌａｇではフラッグ値を累積できるようになっている。また、ＳＵＢＸとＳＵＢのフラッグ変化は、符号拡張／ゼロ拡張を含めてほとんど同じである。ＳＵＢとＳＵＢＸでフラッグ変化の異なるのは、Ｚ＿ｆｌａｇのみである。

(d ≧s の時)

[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] -

[S0.S0.....S0. S0. S1....Ss-2.Ss-1] - X__flag ==>

d-s ビットだけ符号拡張

[R0.R1.....Rd-s-1.Rd-s.Rd-s+1....Rd-2.Rd-1]　(destに設定される)

(d <s の時)

[D0.D0.....D0. D0. D1....Dd-2.Dd-1] -

s-dビットだけ符号拡張

[S0.S1.....Ss-d-1.Ss-d.Ss-d+1....Ss-2.Ss-1] - X__flag ==>

89 90

[P0.P1.....Ps-d-1.Ps-d.Ps-d+1....Ps-2.Ps-1] ==>

_____[R0. R1....Rd-2.Rd-1]（destに設定される）

P0.P1.....Ps-d-1

s-dビットがカットされる。

L__flag☆ S[D] - S[S] - X__flag< 0

符号付きの数と見て演算を行ない、結果が負になることを表わす。d ≠s の場合にはオペランドが符号拡張されてから比較される。

（ M__flagも結果の正負を表わすが、 M__flagが正しい正負を表示するのはオーバーフローのない時に限られる。）

M__flag R0

Z__flag [R0 ～d-1] = 0 .and. previous Z__flag

V__flag S[D] - S[S] - X__flag< -2^(d-1) .or.

S[D] - S[S] - X__flag≧ +2^(d-1)

符号付きの数と見て演算を行ない、結果がオーバーフローすることを表わす。d ≠s の場合にはオペランドが符号拡張される。

X__flag☆ いずれの場合も、destのサイズからの桁下げが X__flag にセットされる。

(d ≧s の時)

U[D0.D1.....Dd-s-1.Dd-s.Dd-s+1....Dd-2.Dd-1] -

U[S0.S0.....S0. S0. S1....Ss-2.Ss-1] -

X__flag< 0

d-s ビットだけ符号拡張

d >s の場合には符号拡張を行なう。これは、destや他のフラッグの設定の処理と共通化したためである。しかし、符号拡張後の演算、比較ではオペランドが符号なしの数として扱われる。

(d <s の時)

U[D0. D1....Dd-2.Dd-1] -

_____U[Ss-d.Ss-d+1 ....Ss-2.Ss-1] -

X__flag< 0

S0.S1.....Ss-d-1

s-dビットがカットされる

【０１２５】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm__data,@SP+,@-SPのとき

〔ニモニック〕

MUL s r c , d e s t

〔命令の機能〕

40 d e s t * s r c ==> d e s t

乗算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図８５に示す。

〔フラッグ変化〕図８６に示す。

〔解説〕ソースオペランドをデスティネーションオペランドに乗ずる。乗算は符号付きで行われ、オペランドも符号付き整数とみなされる。この命令は、被乗数のサイズと結果のサイズが等しいため、高級言語向きである。

50 デスティネーションのサイズが小さく、演算結果がデス

91

ティネーションのサイズの符号付き整数として表現できない時は、Ｖ＿ｆｌａｇがセットされる。オーバーフローが生じた場合にも、ｄｅｓｔにセットされるデータ（正しい結果の下位ビット）が基準となってＭ＿ｆｌａｇ，Ｚ＿ｆｌａｇがセットされる。例えば、Ｒ０＝Ｈ'１００００で

```
MUL.W  #H'10000.R0
```

を実行した場合、積がＨ'１０００００００００となるため、Ｒ０＝０（下位ビット），Ｖ＿ｆｌａｇ＝１，Ｚ＿ｆｌａｇ＝１となる。

【０１２６】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm_data.@SP+,@-SPのとき

〔ニモニック〕

ＭＵＬＵ　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｄｅｓｔ　＊　ｓｒｃ＝＝〉ｄｅｓｔ

符号なし乗算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図８７に示す。

〔フラッグ変化〕図８８に示す。

〔解説〕ソースオペランドをデスティネーションオペランドに乗ずる。乗算は符号なしで行われ、オペランドも＊

92

＊符号なし整数とみなされる。デスティネーションのサイズが小さく、演算結果がデスティネーションのサイズの符号なし整数として表現できない時は、Ｖ＿ｆｌａｇがセットされる。

【０１２７】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm_data.@SP+,@-SPのとき

〔ニモニック〕

ＭＵＬＸ　ｓｒｃ，ｄｅｓｔ，ｔｍｐ

〔命令の機能〕

```
dest * src ==> reg&dest (double size)
```

拡張乗算、サイズが大きくなる

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図８９に示す。

〔フラッグ変化〕図９０に示す。

〔解説〕ソースオペランドをデスティネーションオペランドに乗ずる。この命令は、積が倍長で得られるため、ｓｒｃとｄｅｓｔのほかに、積の上位ビットを入れるためのテンポラリレジスタｔｍｐを指定する。サイズは３２ビットに固定（３２／６４から選択）とする。乗算は符号なしで行われ、積のサイズは被乗数のサイズの２倍になる。

[MULXのオペレーション]

```
dest[0:31] × src[0:31] ==> tmp1[0:63]
tmp1[32:63] ==> tmp[0:31]
tmp1[0:31] ==> dest[0:31]
```

ＭＵＬＸでは、得るべき結果がｄｅｓｔ，ｔｍｐの二つあるので、両者が重なった場合（ｄｅｓｔでｔｍｐと同じレジスタを指定した場合）の処置が問題となる。一般に、ｔｍｐ（ＭＵＬＸの上位桁）の方は次の桁への桁上がりとして用いられることが多いので、最終桁の計算などでは使用しないこともある。したがって、両者が重なった場合には、ｄｅｓｔに設定すべき値（ＭＵＬＸの下位桁）の方が残るものとする。（付録１２参照）

ＭＵＬＸのＭ＿ｆｌａｇ，Ｚ＿ｆｌａｇのフラッグ変化は、ｄｅｓｔを基準とする。ｔｍｐに設定される値は、これらのフラッグには影響しない。このような仕様になったのは、次のような理由による。・ＡＤＤＸ，ＳＵＢＸなどのフラッグ変化の仕様に合わせたため。（ＡＤＤＸ，ＳＵＢＸでは、Ｘ＿ｆｌａｇがセットされていても、ｄｅｓｔが０であればＺ＿ｆｌａｇがセットされる。）

・多倍長演算を考えた場合には、ｔｍｐ＆ｄｅｓｔのみでフラッグを変化させてもあまり意味がない。フラッグ本来の意味で変化させるためには、全体の値を通して判定することが必要であり、個々の命令だけでは対処できない。ｔｍｐ＆ｄｅｓｔでフラッグ変化を行なったとしても、結局中途半端である。

例：

［実行前］

```
R1=H'00000000  dest=H'20000000  src=H'40000000
MULX  @src.@dest.R1
```

［実行後］

【０１２８】

【数６】

93

```
tmp=H' 0800000000000000
      ┆.....┆ ┆.....┆
       R1      dest
```

dest に設定される値が０なので、

Z__flagがセットされる。

【０１２９】なお、ＭＵＬＸ，ＤＩＶＸでは、ＡＤＤＸ，ＳＵＢＸとは異なり、Ｚ__ｆｌａｇは累積した変化＊

94

＊をするわけではない。Ｆ__ｆｌａｇによってｔｍｐ＝０のテストが可能である。！＝０の場合は、動作を保証しない。実際「本発明装置」では！＝０の場合、ｓｒｃサイズを、！Ｒ（８ビットまたは１６ビット）としてオペランドのフェッチを行ない、それを３２ビットに符号拡張して命令が実行される。ただし、ｄｅｓｔ，ｔｍｐは！Ｒによらず常に３２ビットとして扱われる。

【０１３０】〔プログラム例外〕

・予約命令例外

・[R='11' のとき

注）!=0 のときは予約命令例外としては検出しない。

・BaR が@-SPのとき

・BaMRが#imm__data,@SP+,@-SPのとき

〔ニモニック〕

ＤＩＶ　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｄｅｓｔ／ｓｒｃ＝＝〉ｄｅｓｔ

除算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図９１に示す。

〔フラッグ変化〕図９２に示す。

〔解説〕ソースオペランドでデスティネーションオペランドを割る。除算は符号付きで行われ、オペランドも符号付き整数とみなされる。この命令は、被除数のサイズと結果のサイズが等しいため、高級言語向きである。商は０方向に丸められ、余りの符号は被除数と同じになる。

例：

10/3	→	商=3,	剰余=1
(-10)/3	→	商=(-3)，	剰余=(-1)
10/(-3)	→	商=(-3)，	剰余=1

ｓｒｃ＝０の場合には、ゼロ除算例外（ＺＤＥ）となる。ゼロ除算の場合、Ｖ__ｆｌａｇがセットされて例外処理が起動されるが、ｄｅｓｔの値は変化しない。このとき、ｄｅｓｔのライトアクセスを行なうかどうか、すなわち、同じ値を書き込むか、何も書き込まないかは規定しないものとする。また、Ｖ__ｆｌａｇ以外のフラッグも変化しない。これは、ｄｅｓｔに合わせたため、および、例外処理プログラムで例外発生要因を解析するためには、できるだけ以前の状態（フラッグを含めて）が保存されている方が望ましいためである。ＤＩＶで０除算以外の場合にオーバーフローが発生するのは（最小負数）÷（－１）の場合のみである。ＤＩＶはＤＩＶＸとは異なり、コンパイラの生成する普通の演算命令であるため、できるだけ他の演算命令と同じ仕様にする方が望ましい。そこで、この場合のフラッグの変化は、それぞれのフラッグの意味を生かして、

V__flag=1,L__flag=0,M__flag=1,Z__flag=0

{最小負数÷(-1)の時}

とする。オーバーフローするのは（最小負数÷（－１））に限るので、正しい結果の下位ビットがｄｅｓｔにセットされると考えても、結局　ｄｅｓｔは変化しない。正しい結果の下位ビットになったと考えても結局同じ値である。

例：

src=H' ffff=(-1),dest=H' 8000=(-32768)

でDIV.H を実行した場合

==> dest=H' 8000,V__flag=1

destのH' 8000は、正しい結果

(H'...008000=32768) の下位ビットと考えることもできるし

、destが変化しなかったと考えることもできる。

【０１３１】〔プログラム例外〕

95

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm__data,@SP+,@-SPのとき

・ゼロ除算例外

・ｓｒｃ＝０のとき

〔ニモニック〕

ＤＩＶＵ　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｄｅｓｔ／ｓｒｃ==〉ｄｅｓｔ

符号なし除算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図９３に示す。

〔フラッグ変化〕図９４に示す。

〔解説〕ソースオペランドでデスティネーションオペランドを割る。除算は符号なしで行われ、オペランドも符号なし整数とみなされる。ｓｒｃ＝０の場合には、ゼロ除算例外（ＺＤＥ）となる。ゼロ除算の場合、Ｖ__ｆｌａｇがセットされて例外処理が起動されるが、ｄｅｓｔは変化しない。このとき、ｄｅｓｔのライトアクセスを行なうかどうか、すなわち、同じ値を書き込むか、何も書き込まないかは規定しないものとする。また、Ｖ__ｆｌａｇ以外のフラッグも変化しない。これは、ｄｅｓｔに合わせたため、および、例外処理プログラムで例外発生要因を解析するためには、できるだけ以前の状態（フラッグを含めて）が保存されている方が望ましいためである。ＤＩＶＵ命令では、０除算以外の場合に、オーバーフローが発生してＶ__ｆｌａｇがセットされることはない。したがって、０除算以外の場合は必ずＶ__ｆｌａｇがクリアされる。

【０１３２】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm__data,@SP+,@-SPのとき

・ゼロ除算例外

・ｓｒｃ＝０のとき

〔ニモニック〕

ＤＩＶＸ　ｓｒｃ，ｄｅｓｔ，ｔｍｐ

〔命令の機能〕

reg&dest / src ==> dest.reg (quotient.remainder)

拡張除算、サイズが小さくなる、剰余を出す

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図９５に示す。

96

〔フラッグ変化〕図９６に示す。

〔解説〕ソースオペランドでデスティネーションオペランドを割る。この命令は、多倍長除算のプリミティブとなるため、ｓｒｃとｄｅｓｔのほかに、拡張演算のためのテンポラリ値（剰余）を置くレジスタを指定する。サイズは３２ビットに固定（３２／６４から選択）とする。除算は符号なしで行われ、被除数のサイズは除数のサイズの２倍になる。

［ＤＩＶＸのオペレーション］

concatinate(tmp[0:31],dest[0:31]) ==> tmp1[0:63]

quo(tmp1[0:63],src[0:31]) ==> dest[0:31]

rem(tmp1[0:63],src[0:31]) ==> tmp [0:31]

ＤＩＶＸでは、得るべき結果がｄｅｓｔ，ｔｍｐの二つあるので、両者が重なった場合（ｄｅｓｔでｔｍｐと同じレジスタを指定した場合）の処置が問題となる。一般に、ｔｍｐ（ＤＩＶＸの剰余）の方は次の桁への桁下がりとして用いられることが多いので、最終桁の計算などでは使用しないこともある。したがって、両者が重なった場合には、ｄｅｓｔに設定すべき値（ＤＩＶＸの商）の方が残るものとする。なお、ＤＩＶＸは被除数が多倍長の場合に使用できる命令であるが、除数が多倍長になった場合には、ＤＩＶＸが使用できず、プログラムによってシフトや減算を繰り返しながら割り算を進めなければならない。この際、多倍長のシフト演算が必要になる。多倍長のシフトを実現するため、Ｘ__ｆｌａｇを通したローテイト命令（ＳＨＸＲ，ＳＨＸＬ）が用意されている。ＤＩＶＸのＭ__ｆｌａｇ，Ｚ__ｆｌａｇのフラッグ変化は、ｄｅｓｔ（商）を基準とする。ｔｍｐに設定される値（剰余）は、これらのフラッグには影響しない。ただし、Ｆ__ｆｌａｇによってｔｍｐ＝０のテストが可能である。ＭＵＬＸ，ＤＩＶＸではＡＤＤＸ，ＳＵＢＸとは異なり、Ｚ__ｆｌａｇは累積した変化をするわけではない。ＤＩＶＸで結０がオーバーフローした場合、ＭＯＶ，ＡＤＤ，ＳＵＢ，ＭＵＬでのオーバーフローと仕様を合わせるという意味では、正しい結果の下位ビットがｄｅｓｔに設定されるのが望ましい。しかし、ＡＤＤ，ＳＵＢのように、オーバーフローの時も自然に正しい結果の下位ビットが得られるものと違って、除算の場合には上位ビットから計算を行なうため、アルゴリズムの関係で正しい結果の下位ビットを得るのが難しい。したがって、ＤＩＶＸのオーバーフローの場合には、ｄｅｓｔを変化させないという仕様にする。ＤＩＶＸで商がｄｅｓｔに入らず、オーバーフローが発生した場合には、Ｖ__ｆｌａｇ以外のフラッグは変化しない。これは、ＤＩＶＸでオーバーフローが発生した場合に、ｄｅｓｔが変化しないことに合わせたものである。ｓｒｃ＝０の場合には、ゼロ除算例外（ＺＤＥ）となる。ゼロ除算の場合ｄｅｓｔ，ｔｍｐは変化しない。このとき、ｄｅｓｔのライトアクセスを行なうかどうか、すな

97

わち、同じ値を書き込むか、何も書き込まないかは規定しないものとする。また、Ｖ＿ｆｌａｇ以外のフラグも変化しない。これは、ｄｅｓｔに合わせたため、および、例外処理プログラムで例外発生要因を解析するためには、できるだけ以前の状態（フラグを含めて）が保存されている方が望ましいためである。！＝０の場合は、動作を保証しない。実際「本発明装置」では！＝０＊

98

＊の場合、ｓｒｃサイズを、！Ｒ（８ビットまたは１６ビット）としてオペランドのフェッチを行ない、それを３２ビットに符号拡張して命令が実行される。ただし、ｄｅｓｔ，ｔｍｐは！Ｒによらず常に３２ビットとして扱われる。

【０１３３】〔プログラム例外〕

・予約命令例外

・!R='11' のとき

注）!=0 のときは予約命令例外としては検出しない。

・EaR が@-SPのとき

・EaMRが#imm__data,@SP+,@-SPのとき

・ゼロ除算例外

・ｓｒｃ＝０のとき

〔ニモニック〕

ＲＥＭ　　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｄｅｓｔ　％　ｓｒｃ＝＝〉ｄｅｓｔ

剰余

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図９７に示す。

〔フラッグ変化〕図９８に示す。

〔解説〕ソースオペランドでデスティネーションオペランドを割り、その剰余を求める。除算は符号付きで行われ、オペランドも符号付き整数とみなされる。この命令は、被除数のサイズと剰余のサイズが等しいため、高級言語向きである。商は０方向に丸められ、余りの符号は被除数と同じになる。

例：

10/3	→	商=3,	剰余=1
(-10)/3	→	商=(-3)，	剰余=(-1)
10/(-3)	→	商=(-3)，	剰余=1

ｓｒｃ＝０の場合には、ゼロ除算例外（ＺＤＥ）となる。ただし、ＲＥＭで０除算を行なった場合には、オーバーフローをクリアして例外処理を起動するようにする。ＲＥＭ命令では、ＤＩＶ命令とは異なり、０除算をしてもｄｅｓｔ（剰余）がオーバーフローするわけではないので、Ｖ＿ｆｌａｇはクリアしておく方が合理的である。また、Ｖ＿ｆｌａｇをクリアしておくと、例外処理の中でＤＩＶによるエラーかＲＥＭによるエラーかが判定しやすい。０除算の場合、ｄｅｓｔは無変化である。ｄｅｓｔに対してメモリアクセスを行なうか（ｒｅａｄまたは同じ値でｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅ）、アクセスを行なわないか、については、インプリメント方法を縛ることになるので、規定しない。

【０１３４】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm__data,@SP+,@-SPのとき

・ゼロ除算例外

・ｓｒｃ＝０のとき

〔ニモニック〕

ＲＥＭＵ　　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｄｅｓｔ　％　ｓｒｃ＝＝〉ｄｅｓｔ

符号なし除算による剰余

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図９９に示す。

〔フラッグ変化〕図１００に示す。

〔解説〕ソースオペランドでデスティネーションオペランドを割り、その剰余を求める。除算は符号なしで行われ、オペランドも符号なし整数とみなされる。ｓｒｃとｄｅｓｔのサイズの違う場合にはゼロ拡張が行なわれる。この命令は、被除数のサイズと剰余のサイズが等しいため、高級言語向きである。ｓｒｃ＝０の場合には、ゼロ除算例外（ＺＤＥ）となる。０除算の場合の処置はＲＥＭと同様である。

【０１３５】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm__data,@SP+,@-SPのとき

・ゼロ除算例外

・ｓｒｃ＝０のとき

〔ニモニック〕

ＮＥＧ　　ｄｅｓｔ

〔命令の機能〕

99

０－ｄｅｓｔ＝＝〉ｄｅｓｔ
補数演算
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１０１に示す。

100

〔フラッグ変化〕図１０２に示す。
〔解説〕オペランドの符号を反転する。

L__flag　命令実行後のdestの値が負のとき、すなわちdestの初期値が正のときセットされる。
M__flag　命令実行後のdestのMSB が1 のとき、すなわちdestの初期値が正または最小負数のときセットされる。
Z__flag　命令実行後のdestの値が0 のとき、すなわちdestの初期値が0 のときセットされる。
V__flag　destの初期値が最小負数（MSB のみ1 で他のビットはall 0 ）のみセットされる。

【０１３６】〔プログラム例外〕
・予約命令例外
・MM='11' のとき
・EaM が#imm__data,@SP+,@-SPのとき

〔ニモニック〕
ＩＮＤＥＸ　ｉｎｄｅｘｓｉｚｅ，ｓｕｂｓｃｒｉｐｔ，ｘｒｅｇ
〔命令の機能〕
ｃａｌｃｕｌａｔｅ　ａｄｄｒｅｓｓ　ｏｆ　ａｒｒａｙ
多次元配列のアドレス計算
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１０３に示す。
〔フラッグ変化〕図１０４に示す。
〔解説〕多次元配列を一次元配列に展開するためのアドレス計算として、スケールの乗算とインデクスの加算を行なう。ｓｕｂｓｃｒｉｐｔのサイズがｘｒｅｇのサイズよりも小さい時は、ｓｕｂｓｃｒｉｐｔが符号拡張される。ｘｒｅｇ，ｉｎｄｅｘｓｉｚｅ，ｓｕｂｓｃｒｉｐｔは符号付きの整数と考え、乗算および加算は符号付きで行なう。乗算または加算でオーバーフローを検出した場合には、Ｖ__ｆｌａｇをセットする。ｉｎｄｅｘｓｉｚｅは固定値（ｉｍｍｅｄｉａｔｅ）で済むことが多いが、メモリ上に配列のディスクリプタを作ることを考え、汎用アドレッシングとしている。ＩＮＤＥＸ命令がＣＨＫ命令の後で実行されるならば、ｓｕｂｓｃｒｉｐ※ｔはレジスタのみの指定でよい。しかし、高級言語の仕様によっては範囲をチェックしない（ＣＨＫ命令を実行しない）こともあるので、メモリ上の変数をｓｕｂｓｃｒｉｐｔとして使用できるように、ｓｕｂｓｃｒｉｐｔも汎用アドレッシングとしている。
［ＩＮＤＥＸのオペレーション］

```
xreg * indexsize + subscript ==> xreg
```

ＩＮＤＥＸ命令では、オペランドｘｒｅｇ，ｉｎｄｅｘｓｉｚｅ，ｓｕｂｓｃｒｉｐｔは、すべてポインタではなく符号付きの数として扱われる。負であってもそのまま演算し、ＥＩＴなどの特別な動作はしない。また、フラッグ変化（Ｖ__ｆｌａｇ，Ｌ__ｆｌａｇ，Ｍ__ｆｌａｇ，Ｚ__ｆｌａｇ）も一般の算術演算命令に準じたものになっている。これに関して若干補足しておく。ＩＮＤＥＸで扱うオペランドは、ポインタというよりも配列のインデクスであり、ＩＮＤＥＸでは配列のインデクスを一次元に展開するという処理を行なう。インデクスがポインタになるのは、付加モードのスケーリングで（×４）などを行なった後である。したがって、ＩＮＤＥＸで扱うデータを符号付きと考えても特に不自然はないし、インデクスが負になると困るような言語では、それをチェックすることもできる。！＝０のときは、動作を保証しない。実際「本発明装置」では！＝０の場合、ｉｎｄｅｘｓｉｚｅサイズを！Ｒ（８ビットまたは１６ビット）としてオペランドのフェッチを行ない、それを３２ビットに符号拡張して命令が実行される。ただし、ｘｒｅｇは！Ｒによらず常に３２ビットとして扱われる。
【０１３７】〔プログラム例外〕
・予約命令例外
・!R='11' のとき
・SS='11' のとき
注）!='0' のときは予約命令例外としては検出しない。
・EaR,EaR2が@-SPのとき

【０１３８】１２－４．論理演算命令
〔ニモニック〕
ＡＮＤ　ｓｒｃ，ｄｅｓｔ
〔命令の機能〕

101

ｄｅｓｔ．ａｎｄ．ｓｒｃ＝＝〉ｄｅｓｔ
論理積
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１０５に示す。
〔フラッグ変化〕図１０６に示す。
〔解説〕ソースオペランドとデスティネーションオペランドとの論理積をとる。ソースオペランドのサイズがデスティネーションオペランドのサイズと異なる異種サイズ間の演算（ＡＮＤ：ＧのＲＲ≠ＭＭ、ＡＮＤ：ＥのＭＭ≠００）になった場合、命令はそのまま実行され、予約命令例外とはならないが、ｄｅｓｔに設定される結果＊

・予約命令例外
・RR='11' のとき
・MM='11' のとき
・EaR が@-SPのとき
・EaM が#imm_data,@SP+,@-SPのとき

102

＊は保証できない（インプリメント依存になる）ものとする。「本発明装置」の仕様としては、異種サイズ間の論理演算を規定していないので、この機能をソフトウエアで利用してはいけない。異種サイズ間の論理演算はあまり意味のない命令であるが、これを予約命令例外としないのは、インプリメントの負担が大きく、実行速度に影響が出るためである。

M_flag　R0
Z_flag　[R0 ～d-1] = 0

【０１３９】〔プログラム例外〕

〔ニモニック〕
ＯＲ　ｓｒｃ，ｄｅｓｔ
〔命令の機能〕
ｄｅｓｔ．ｏｒ．ｓｒｃ＝＝〉ｄｅｓｔ
論理和
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１０７に示す。
〔フラッグ変化〕図１０８に示す。
〔解説〕ソースオペランドとデスティネーションオペランドとの論理和をとる。ソースオペランドのサイズがデスティネーションオペランドのサイズと異なる異種サイズ間の演算（ＯＲ：ＧのＲＲ≠ＭＭ、ＯＲ：ＥのＭＭ≠００）になった場合、命令はそのまま実行され、予約命令例外とはならないが、ｄｅｓｔに設定される結果は保証できない（インプリメント依存になる）ものとする。

M_flag　R0
Z_flag　[R0 ～d-1] = 0

【０１４０】〔プログラム例外〕
・予約命令例外
・RR='11' のとき
・MM='11' のとき
・EaR が@-SPのとき
・EaM が#imm_data,@SP+,@-SPのとき

〔ニモニック〕
ＸＯＲ　ｓｒｃ，ｄｅｓｔ
〔命令の機能〕
ｄｅｓｔ．ｘｏｒ．ｓｒｃ＝＝〉ｄｅｓｔ
排他的論理和
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１０９に示す。
〔フラッグ変化〕図１１０に示す。
〔解説〕ソースオペランドとデスティネーションオペランドとの排他的論理和をとる。ソースオペランドのサイズがデスティネーションオペランドのサイズと異なる異種サイズ間の演算（ＸＯＲ：ＧのＲＲ≠ＭＭ、ＸＯＲ：ＥのＭＭ≠００）になった場合、命令はそのまま実行され、予約命令例外とはならないが、ｄｅｓｔに設定される結果は保証できない（インプリメント依存になる）ものとする。

M_flag　R0
Z_flag　[R0 ～d-1] = 0

【０１４１】〔プログラム例外〕
・予約命令例外
・RR='11' のとき
・MM='11' のとき
・EaR が@-SPのとき
・EaM が#imm_data,@SP+,@-SPのとき

〔ニモニック〕
ＮＯＴ　ｄｅｓｔ
〔命令の機能〕
【０１４２】
【数７】

$$-dest \rightarrow dest$$

【０１４３】全ビット反転
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１１１に示

103

す。

〔フラッグ変化〕図１１２に示す。

M__flag　命令実行後のdestのMSB が1 のとき、すなわちdestの初期値のMSB が0 のときセットされる。

Z__flag　命令実行後のdestの値が0 のとき、すなわちdestの初期値が0 のときセットされる。

【０１４４】〔プログラム例外〕

・予約命令例外

・MM='11' のとき

・EaM が#imm__data,@SP+,@-SPのとき

【０１４５】１２－５．シフト命令

〔ニモニック〕

ＳＨＡ　ｃｏｕｎｔ，ｄｅｓｔ

〔命令の機能〕

ｓｈｉｆｔ　ａｒｉｔｈｍｅｔｉｃ

算術シフト

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１１３に示す。

〔フラッグ変化〕図１１４に示す。

〔解説〕デスティネーションオペランドｄｅｓｔをソースオペランドｃｏｕｎｔで指定されたビット数だけ算術シフトする。一般形の命令では、ｃｏｕｎｔの符号によってシフト方向を指定する。ｃｏｕｎｔが正の時左シフト、負の時右シフトとなる。算術シフトであるため、右シフトの場合にはデスティネーションのＭＳＢ（符号ビット）が変化せず、同じ値が右側のビットにコピーされていく。左シフトの場合には、ＬＳＢに０が入り、０が左側のビットにコピーされていく。正負によるシフト方向の指定は、浮動小数点演算のエミュレーションなどに有効な場合がある。左シフトの場合はＳＨＡの短縮形がないが、フラッグ変化がＳＨＡと異なってもよければ、ＳＨＬの短縮形ＳＨＬ：Ｑで代用できる。

〔左シフト（ｃｏｕｎｔ＞０）の場合〕図１１５に示す。

〔右シフト（ｃｏｕｎｔ＜０）の場合〕図１１６に示す。

なお、ｃｏｕｎｔ＝０の場合は、Ｘ__ｆｌａｇ＝０となる。ＳＨＡ命令では、ｃｏｕｎｔのサイズとして８ビットのみが有効である。ＲＲ≠００の場合動作を保証しない。ＲＲ≠００の機能が利用できないのは、主としてインプリメント上の制約によるものである。ＲＲ≠００の場合には、「本発明装置」では、サイズＲＲでｃｏｕｎｔオペランドのフェッチを行ない、ｃｏｕｎｔの下位８ビットのみを有効としてそのまま命令を実行する。ＳＨＡは算術演算なのでＬ__ｆｌａｇをセットするが、これはｄｅｓｔのはじめの値の符号（ＭＳＢ）を反映する。これは、Ｌ__ｆｌａｇが、オーバーフローやアンダーフローが発生した場合にも、常に正しい計算結果の符号を反映するという性質を持っているためである。シフト命令の場合、オーバーフローが起こらなければ、ｄｅｓｔの符号は変化しない。右シフトの場合、または左シフトでオーバーフローがなかった場合にはＬ__ｆｌａｇ＝Ｍ__ｆｌａｇとなるが、左シフトでオーバーフローが発生した場合には、Ｌ__ｆｌａｇとＭ__ｆｌａｇが異なる変化をすることがある。「本発明装置」はｂｉｇ－ｅｎｄｉａｎのチップであるため、ｃｏｕｎｔをビット位置の増減の意味で考えるか、２の累乗の意味で考えるかによって、シフト方向が逆になってしまう。すなわち、前者の考え方ではｃｏｕｎｔ＞０の時右シフトとすべきであるのに対して、後者の考え方では、ｌｉｔｔｌｅ－ｅｎｄｉａｎの場合と同じように、ｃｏｕｎｔ＞０の時左シフトとなる。しかし、シフト命令はビット操作関係の命令よりも算術演算関係の命令に近いものであるから、ｃｏｕｎｔをビット位置の増減の意味で考えるよりは、２の累乗の意味で考える方が自然である。したがって、「本発明装置」ではｃｏｕｎｔ＞０の時左シフトという仕様になっている。ＳＨＬ，ＳＨＡでは、ｃｏｕｎｔの絶対値が（ｄｅｓｔのサイズ＋１）を越えた場合にも、指定された数だけそのままシフトを続ける。結果的に、ｃｏｕｎｔの絶対値が（ｄｅｓｔのサイズ＋１）の場合と同じ動作になる。例えば、次のような動作をする。

```
SHA     #33,dset.W      ;dest=X__flag=0
SHL     #33,dest.W      ;dest=X__flag=0
SHA     #-33,dest.W     ;dest=X__flag= 旧destのMSB
SHL     #-33,dest.W     ;dest=X__flag=0
```

なお、Ｘ__ｆｌａｇを除けば、ｃｏｕｎｔの絶対値が（ｄｅｓｔのサイズ）に等しい場合も同じ結果になる。

【０１４６】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM,ShM が#imm__data,@SP+,@-SPのとき

〔ニモニック〕

ＳＨＬ　ｃｏｕｎｔ，ｄｅｓｔ

〔命令の機能〕

ｓｈｉｆｔ　ｌｏｇｉｃａｌ

論理シフト

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１１７に示す。

〔フラッグ変化〕図１１８に示す。

〔解説〕デスティネーションオペランドをソースオペランドｃｏｕｎｔで指定されたビツト数だけ論理シフトする。一般形の時、シフト方向はｃｏｕｎｔの符号で指定する。ｃｏｕｎｔが正の時左シフト、負の時右シフトとなる。右シフトの場合には、ＭＳＢに０が入り、０が右側のビットにコピーされていく。また、左シフトの場合には、ＬＳＢに０が入り、０が左側のビットにコピーされていく。

〔左シフト（ｃｏｕｎｔ＞０）の場合〕図１１９に示す。

〔右シフト（ｃｏｕｎｔ＜０）の場合〕図１２０に示す。

なお、ｃｏｕｎｔ＝０の場合は、Ｘ＿ｆｌａｇ＝０となる。ＳＨＬ命令では、ｃｏｕｎｔのサイズとして８ビットのみが有効である。ＲＲ≠００の場合、動作を保証しない。ＲＲ≠００の機能が利用できないのは、主としてインプリメント上の制約によるものである。ＲＲ≠００の場合には、「本発明装置」ではサイズＲＲでｃｏｕｎｔオペランドのフェッチを行ない、ｃｏｕｎｔの下位８ビットのみを有効としてそのまま命令を実行する。

【０１４７】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM.ShM が#imm__data.@SP+.@-SPのとき

〔ニモニック〕

ＲＯＴ　ｃｏｕｎｔ，ｄｅｓｔ

〔命令の機能〕

ｒｏｔａｔｅ

回転

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１２１に示す。

〔フラッグ変化〕図１２２に示す。

〔解説〕デスティネーションオペランドをソースオペランドｃｏｕｎｔで指定されたビット数だけ回転する。すなわち、ＬＳＢ（ＭＳＢ）から溢れたビットをＭＳＢ（ＬＳＢ）に詰めながらシフトを行なう。回転方向はｃｏｕｎｔの符号で指定する。ｃｏｕｎｔが正の時左回転、負の時右回転となる。回転の際、フラッグは通さない。

〔左回転（ｃｏｕｎｔ＞０）の場合〕図１２３に示す。

〔右回転（ｃｏｕｎｔ＜０）の場合〕図１２４に示す。

なお、ｃｏｕｎｔ＝０の場合は、Ｘ－ｆｌａｇ＝０となる。ＲＯＴ命令では、ｃｏｕｎｔのサイズとして８ビットのみが有効である。ＲＲ≠００の場合、動作を保証しない。ＲＲ≠００の機能が利用できないのは、主としてインプリメント上の制約によるものである。ＲＲ≠００の場合には、「本発明装置」ではサイズＲＲでｃｏｕｎｔオペランドのフェッチを行ない、ｃｏｕｎｔの下位８ビットのみを有効としてそのまま命令を実行する。ＲＯＴでｃｏｕｎｔの絶対値が（ｄｅｓｔのサイズ）を越えた場合にも、指定された数だけそのままローテイトを続ける。結果的に、ｃｏｕｎｔを（ｄｅｓｔのサイズ）で割った剰余をｃｏｕｎｔとした場合と同じ動作になる。ただし、ｃｏｕｎｔが（ｄｅｓｔのサイズ）の整数倍（≠０）の場合には、ＭＳＢまたはＬＳＢに応じてＸ＿ｆｌａｇのセットされる点が、ｃｏｕｎｔ＝０の場合とは異なる。例えば、データサイズと同じビット数だけ回転した場合、データの値そのものは変化せず、ｄｅｓｔはｃｏｕｎｔ＝０の時と同じ値になる。しかし、フラッグにはもとのデータのＬＳＢがコピーされるため、フラッグ変化はｃｏｕｎｔ＝０の時とは異なったものになる。

【０１４８】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm__data.@SP+.@-SPのとき

〔ニモニック〕

ＳＨＸＬ　ｄｅｓｔ

〔命令の機能〕

ｓｈｉｆｔ　ｌｅｆｔ　ｗｉｔｈ　ｅｘｔｅｎｄ

拡張左シフト

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１２５に示す。

〔フラッグ変化〕図１２６に示す。

〔解説〕ｄｅｓｔを１ビット左にシフトし、元のＸ　ｆｌａｇの内容をＬＳＢに詰める。ＭＳＢからあふれたビットはＸ　ｆｌａｇに入る。この命令は、複数ワードの１ビットシフトを行なうためのプリミティブとして専用化したものであり、シフト対象のサイズを３２ビットに固定している点、１ビットのシフトしかできない点、などにおいて　ＳＨＡ，ＳＨＬ，ＲＯＴとはかなり仕様が異なる。ＤＩＶＸは被除数が多倍長の場合に使用できる命令であるが、除数が多倍長になった場合には、ＤＩＶＸが使用できず、プログラムによってシフトや減算を繰り返しながら割り算を進めなければならない。その際、多倍長のシフト演算が必要になる。この命令は、このような場合に使用することを目的とした命令である。図１

107

２７にこれを示す。

【０１４９】〔プログラム例外〕

・予約命令例外

・+='0' のとき

・-='1' のとき

・X='1' のとき

・EaMXが#imm__data,@SP+,@-SPのとき

〔ニモニック〕

ＳＨＸＲ　ｄｅｓｔ

〔命令の機能〕

ｓｈｉｆｔ　ｒｉｇｈｔ　ｗｉｔｈ　ｅｘｔｅｎｄ

拡張右シフト

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１２８に示す。

〔フラッグ変化〕図１２９に示す。

〔解説〕ｄｅｓｔを１ビット右にシフトし、元のＸ　ｆｌａｇの内容をＭＳＢに詰める。ＬＳＢからあふれたビットはＸ　ｆｌａｇに入る。この命令は、複数ワードの１ビットシフトを行なうためのプリミティブとして専用化したものであり、シフト対象のサイズを３２ビットに固定している点、１ビットのシフトしかできない点、などにおいて　ＳＨＡ，ＳＨＬ，ＲＯＴとはかなり仕様が異なる。ＤＩＶＸは被除数が多倍長の場合に使用できる命令であるが、除数が多倍長になった場合には、ＤＩＶＸが使用できず、プログラムによってシフトや減算を繰り返しながら割り算を進めなければならない。その際、多倍長のシフト演算が必要になる。この命令は、このよ＊

108

＊うな場合に使用することを目的とした命令である。図１３０にこれを示す。

【０１５０】〔プログラム例外〕

・予約命令例外

・+='0' のとき

・-='1' のとき

・X='1' のとき

・EaMXが#imm__data,@SP+,@-SPのとき

10

〔ニモニック〕

ＲＶＢＹ　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｒｅｖｅｒｓｅ　ｂｙｔｅ　ｏｒｄｅｒ

バイト順の逆転

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１３１に示す。

〔フラッグ変化〕図１３２に示す。

20 〔解説〕ｓｒｃのバイト順を逆転したものをｄｅｓｔに転送する。ｓｒｃよりもｄｅｓｔのサイズの方が大きい場合には、ｓｒｃをｄｅｓｔのサイズにまでゼロ拡張した後、ｄｅｓｔのサイズでバイト順を逆転する。ｓｒｃよりもｄｅｓｔのサイズの方が小さい場合には、ｓｒｃの上位バイトをカットしてｄｅｓｔのサイズとした後、ｄｅｓｔのサイズでバイト順を逆転する。（ｓｒｃのアドレスをずらした上でｓｒｃとｄｅｓｔを同じサイズとしても、結果は同じになる）

例：

```
src = H'1234
RVBY    src.H, dest.H ==> dest = H'3412
RVBY    src.H, dest.W ==> dest = H'34120000
RVBY    src.H, dest.B ==> dest = H'34  (H'12ではない)
```

この命令は、ｅｎｄｉａｎの変換に対するオーバーヘッドを削減することを目的とした命令である。

【０１５１】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・WW='11' のとき

・EaR が@-SPのとき

・EaW が#imm__data,@SP+ のとき

〔ニモニック〕

ＲＶＢＩ　ｓｒｃ，ｄｅｓｔ

但し〈〈Ｌ２〉〉

〔命令の機能〕

ｒｅｖｅｒｓｅ　ｂｉｔ　ｏｒｄｅｒ

ビット順の逆転

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１３３に示す。

〔フラッグ変化〕図１３４に示す。

40 〔解説〕ｓｒｃのビット順を逆転したものをｄｅｓｔに転送する。ｓｒｃよりもｄｅｓｔのサイズの方が大きい場合には、ｓｒｃをｄｅｓｔのサイズにまでゼロ拡張した後、ｄｅｓｔのサイズでビット順を逆転する。ｓｒｃよりもｄｅｓｔのサイズの方が小さい場合には、ｓｒｃの上位バイトをカットしてｄｅｓｔのサイズとした後、ｄｅｓｔのサイズでビット順を逆転する。（ｓｒｃのアドレスをずらした上でｓｒｃとｄｅｓｔを同じサイズとしても、結果は同じになる）

この命令は、ｅｎｄｉａｎの変換に対するオーバーヘッドを削減することを目的とした命令である。ビットマッ

プの処理を考えると、ビットを逆順にするビットリバース命令ＲＶＢＩもあった方がよいが、バイトリバース命令ＲＶＢＹよりも頻度が少ないこと、追加のハードウエ＊

＊アが必要となる可能性が強いこと、により、ＲＶＢＩ命令は〈〈Ｌ２〉〉とする。

【０１５２】〔プログラム例外〕

- 予約命令例外
- RR='11' のとき
- WW='11' のとき
- EaR が@-SPのとき
- EaW が#imm_data.@SP+ のとき

【０１５３】１２－６．ビット操作命令

「本発明装置」のビット操作命令では、次の図のようにｂａｓｅ（ｂａｓｅ　ａｄｄｒｅｓｓ）ｏｆｆｓｅｔ（ｂｉｔ　ａｄｄｒｅｓｓ）の２つのパラメータにより操作対象となるビットを指定する。レジスタ上のビットを操作する場合には、このほかにｂａｓｅのサイズも対象ビットの指定に影響する。［メモリ上のビット操作をする場合］図１３５に示す。「本発明装置」の一般形のビット操作命令では、ｏｆｆｓｅｔの値に制限がなく、ｏｆｆｓｅｔがバイト境界を越えてもよいようになっている。ｏｆｆｓｅｔは、符号付き整数として扱われる。ビット操作命令では、ＢＢフィールドによりメモリアクセスの範囲を指定できるようになっている。これは、ＢＴＳＴでｒｅａｄを行なうメモリアドレスの範囲、およびＢＳＥＴ，ＢＣＬＲ，ＢＮＯＴで　ｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅを行なうメモリアドレスの範囲を意味するものである。アクセスされるメモリアドレスの範囲は、入出力やマルチプロセッサを使用した場合に問題となることがある。しかし、実際には、バイト単位のアクセス（'．Ｂ'）のみが使用できればほとんどの場合に十分であるため、ハーフワード、ワード単位のアクセスは〈〈Ｌ２〉〉とする（ただしレジスタに対するビット操作命令を除く）。また、ハーフワード、ワード単位のアクセスが意味を持つのは、アラインメントのとれたハーフワードやワードをアクセスする場合に限られているので、ハーフワード、ワード単位のアクセスの機能を利用するためには、ｂａｓｅとして必ずアラインメントのとれたアドレスを指定しなければならないという制限を設ける。これは、アクセス範囲の指定に関するインプリメントを容易にするためである。したがって、アラインメントのとれたハーフワードの単位で、対象ビットを含むメモリアクセスを行ないたい場合には、ｂａｓｅとして２の倍数を指定する必要がある。また、アラインメントのとれたワードの単位で、対象ビットを含むメモリアクセスを行ないたい場合には、ｂａｓｅとして４の倍数を指定する必要がある。ｏｆｆｓｅｔの値については制限はない。ｂａｓｅとしてアラインメントのとれていないアドレスを指定した場合のアクセス範囲は、インプリメントに依存するものとする。「本発明装置」では、〈〈Ｌ２〉〉となっているメモリに対するハーフワード、ワード単位のアクセスのインプリメントを行なう。またｂａｓｅとしてアラインメントのとれていないアドレスを指定した場合にも、アクセス範囲はアラインメントのとれたハーフワード、ワード単位でアクセスを行なう。

111　　112

[例]

BSET.B #H'84,@H'100

offset % 8 = 4; base + offset/8 = H'110 なので、H'110 のbit4をセットする。

H'110 の1 バイトについてread-modify-write が行なわれる。

BSET.B #H'7c,@H'101

offset % 8 = 4; base + offset/8 = H'110 でアクセスサイズがバイトなので、BSET.B #H'84,@H'100 と全く同じ動作をする。

BSET.W #H'84,@H'100

offset % 8 = 4; base + offset/8 = H'110 なので、H'110 のbit4をセットする。

baseが４の倍数になっているので、H'100 ～H'103 のアラインメントのとれた32ビットについてread-modify-write を行ない、対象ビットをセットする。

BSET.W #H'7c,@H'101

offset % 8 = 4 ;base + offset/8 = H'110 なので、おなじようにH'110 のbit4をセットする。

しかし、baseが４の倍数ではないので、read-modify-write を行なうアクセス範囲についてはインプリメント依存である。

なお、ＢＢで示されるサイズは、「どの範囲に対してｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅを行なうか？」ということであり、オフセットの範囲（例えば、'．Ｂ'であればオフセットが８より小さくなる、など）を規定するものではない。レジスタに対するビット操作命令では、アクセスのサイズ（ｂａｓｅのサイズ）によってｏｆｆｓｅｔ＝０（ＭＳＢ）のビット位置が変わるため、ｂａｓｅのサイズは重要な意味を持つ。ｂａｓｅがレジスタ直接Ｒｎの場合は、ｂａｓｅのサイズ'．H'，'．W'も〈〈Ｌ１〉〉である。レジスタＲｎをｂａｓｅとしたビット操作命令の場合、ｏｆｆｓｅｔは'．Ｂ'の時下位３ビット、'．Ｈ'の時下位４ビット、'．Ｗ'の時下位５ビット、'．Ｌ'の時下位６ビットのみが有効であり、上位ビットは無視される。上位ビットが０でなくても、エラー、ＥＩＴとはしない。ア＊

＊ーキテクチャ面から見ると、上位ビットを無視するよりも、ＢＦ命令のｗｉｄｔｈなどと同じように、きちんとｏｆｆｓｅｔの範囲をチェックする方が望ましいが、チェックを行なうことにより命令の実行時間が増大するため、ｏｆｆｓｅｔはアクセスサイズのビット数でｍｏｄｕｌｏをとって使用することにしている。レジスタ上に８ビットデータ、１６ビットデータ、３２ビットデータを置いた場合では、それぞれのデータで同じビット位置を持つビットであっても、実際には異なったビットに対応することになるので、注意が必要である。仕様が複雑化するのを避けるため、アセンブラのデフォルトはメモリ対象、レジスタ対象とも．Ｂとする。また、短縮形も．Ｂの仕様である。したがって、短縮形でアクセスできるレジスタ上の範囲は、２＾０～２＾７のビットである（図１３６参照）。

[例]　BSET:Q #1,r0では、

BSETの場合にデフォルトが.Bなので、

r0.Bのビット１がセットされる。

このビットは、r0.Wのビット１とは異なったものであり、

r0.Wのビット25に対応する。

例えば、２＾１７のビットをアクセスするつもりで

ＢＴＳＴ　＃１４，Ｒ０

と書くと、実際は

ＢＴＳＴ．Ｂ　＃１４，Ｒ０

と解釈され、ｏｆｆｓｅｔは上位ビットを無視するので、結局２＾１のビットが対象となる。正しくは、

ＢＴＳＴ．Ｗ　＃１４，Ｒ０

と書かなければならない。このような場合には、アセンブラで警告を出すのが望ましいであろう。

〔ニモニック〕

ＢＴＳＴ　ｏｆｆｓｅｔ，ｂａｓｅ

〔命令の機能〕

113

【０１５４】
【数８】

```
-bit -> Z__flag
```

【０１５５】ビットのテスト
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１３７に示す。
〔フラッグ変化〕図１３８に示す。
〔解説〕指定されたビットの値を反転したものをＺ＿ｆｌａｇにコピーする。ＥａＲｆ，ＳｈＲｆｑで指定されるアドレッシングモードでは、イミディエートモード＃ｉｍｍ＿ｄａｔａ、@－ＳＰ、@ＳＰ＋は使用できない。また、Ｒｎのモードを使用した場合、ｏｆｆｓｅｔの上位ビットの値は無視される。アセンブラ表記では、メモリアクセスのサイズをｂａｓｅのサイズとして指定する。ＢＴＳＴ：Ｑでは、メモリアクセスのサイズは８ビットに固定されており、サイズは’．Ｂ’のみを書くことができる。
【０１５６】〔プログラム例外〕
・予約命令例外
・RR='11' のとき
・BB='11' のとき
・EaR が@-SPのとき
・EaRf, ShRfqが#imm__data, @SP+, @-SPのとき

〔ニモニック〕
ＢＳＥＴ　ｏｆｆｓｅｔ，ｂａｓｅ
〔命令の機能〕
【０１５７】
【数９】

```
-bit -> Z__flag.1 -> bit
```

【０１５８】ビットのセット
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１３９に示す。
〔フラッグ変化〕図１４０に示す。
〔解説〕指定されたビットの値を反転したものをＺ＿ｆｌａｇにコピーし、その後そのビットを１にセットする。ＥａＭｆ，ＳｈＭｆｑで指定されるアドレッシングモードでは、イミディエートモード＃ｉｍｍ＿ｄａｔａ、@－ＳＰ，@ＳＰ＋は使用できない。また、Ｒｎのモードを使用した場合、ｏｆｆｓｅｔの上位ビットの値は無視される。アセンブラ表記では、メモリアクセスのサイズをｂａｓｅのサイズとして指定する。ＢＳＥＴ：Ｑでは、メモリアクセスのサイズは８ビットに固定されており、サイズは’．Ｂ’のみを書くことができる。
【０１５９】〔プログラム例外〕

114

・予約命令例外
・RR='11' のとき
・BB='11' のとき
・EaR が@-SPのとき
・EaMf, ShMfqが#imm__data, @SP+, @-SPのとき

〔ニモニック〕
ＢＣＬＲ　ｏｆｆｓｅｔ，ｂａｓｅ
〔命令の機能〕
【０１６０】
【数１０】

```
-bit -> Z__flag.0 -> bit
```

【０１６１】ビットのクリア
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１４１に示す。
〔フラッグ変化〕図１４２に示す。
〔解説〕指定されたビットの値を反転したものをＺ＿ｆｌａｇにコピーし、その後そのビットを０にクリアする。ＥａＭｆ，ＳｈＭｆｑで指定されるアドレッシングモードでは、イミディエートモード＃ｉｍｍ＿ｄａｔａ、@－ＳＰ，@ＳＰ＋は使用できない。また、Ｒｎのモードを使用した場合、ｏｆｆｓｅｔの上位ビットの値は無視される。アセンブラ表記では、メモリアクセスのサイズをｂａｓｅのサイズとして指定する。ＢＣＬＲ：Ｑでは、メモリアクセスのサイズは８ビットに固定されており、サイズは’．Ｂ’のみを書くことができる。
【０１６２】〔プログラム例外〕
・予約命令例外
・RR='11' のとき
・BB='11' のとき
・EaR が@-SPのとき
・EaMf, ShMfqが#imm__data, @SP+, @-SPのとき

〔ニモニック〕
ＢＮＯＴ　ｏｆｆｓｅｔ，ｂａｓｅ
〔命令の機能〕
【０１６３】
【数１１】

```
-bit -> Z__flag.  -bit -> bit
```

【０１６４】ビットの反転
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１４３に示す。
〔フラッグ変化〕図１４４に示す。
〔解説〕指定されたビットの値を反転したものをＺ＿ｆｌａｇにコピーし、その後そのビットも反転する。ＥａＭｆで指定されるアドレッシングモードでは、イミディエートモード＃ｉｍｍ＿ｄａｔａ、@－ＳＰ，@ＳＰ＋は使用できない。また、Ｒｎのモードを使用した場合、

115

ｏｆｆｓｅｔの上位ビットの値は無視される。アセンブラ表記では、メモリアクセスのサイズをｂａｓｅのサイズとして指定する。

【０１６５】〔プログラム例外〕

・予約命令例外

・RB='11' のとき

・BB='11' のとき

・EaR が@-SPのとき

・EaMfが#imm_data,@SP+,@-SPのとき

116

＊〔ニモニック〕

ＢＳＣＨ　ｄａｔａ，ｏｆｆｓｅｔ

〔命令の機能〕

＊10

find first '0' or '1' in the bitfield(within a word)

０または１のサーチ（１ワード内）

〔命令オプション〕

/0　'0' をサーチ（デフォルト）

/1　'1' をサーチ

/F　ビット番号の増加方向にサーチ（デフォルト）

/B　ビット番号の減少方向にサーチ

<<L2>>　(M32 でサポートする)

〔命令ビットパターンとアセンブラ表記〕図１４５に示す。

〔フラッグ変化〕図１４６に示す。

〔解説〕ワード中にある’０’　または’１’のビットをサーチする。サーチを開始するビット番号（ｂｉｔ　ｏｆｆｓｅｔ）をｏｆｆｓｅｔオペランドにセットしてこの命令を実行すると、命令実行後、サーチ結果のビット番号がｏｆｆｓｅｔオペランドにセットされている。ｏｆｆｓｅｔはｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅとなっている。ｏｆｆｓｅｔをｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅの形にしたのは、ビットの検索を繰り返して行なうことを想定したためである。サーチの行なわれるビット位置の範囲は、ｄａｔａオペランドの０～（ｄａｔａのサイズ）に限られており、ワード境界を越えることはできない。ｏｆｆｓｅｔとしてはすべてのサイズが使用可能であるが、ｏｆｆｓｅｔの初期値の上位ビットはサーチの際には無視される。これは、上位ビットに、ワード境界を越えた分のビットオフセットや実効アドレスなどといった、別の情報が入っていることが多いと考えられるためである。また、ＢＳＣＨの仕様を軽くして高速化するためである。なお、／Ｆの時ｄａｔａのサイズを表わす数、／Ｂの時（－１）となる。「上位ビット」とは、ｌｏｇ２（ｄａｔａのビット数）よりも上位のビットを示す。ｄａｔａが３２ビットであれば、２＾５～２＾３１のビットが上位ビットである。サーチはビット番号の大きな方向へ向かって、つまり、ｂｉｇ－ｅｎｄｉａｎの「本発明装置」では　ＬＳＢの方向へ向かって行なわれるのが標準の仕様〈〈Ｌ０〉〉であり、／Ｆオプションにより実現される。逆方向のサーチ／Ｂオプションは〈〈Ｌ２〉〉仕様となっている。これは、正方向のサーチと逆方向のサーチで全く別のハードウエアが必要なためである。また、サーチされるｄａｔａのサイズのうち、８ビットと１６ビット（ＲＲ＝００，０１）の指定は〈〈Ｌ２〉〉となっている。「本発明装置」では、〈〈Ｌ２〉〉となっている／Ｂオプション，８ビットと１６ビットのデータサイズ（ＲＲ＝００，０１）もサポートする。ＢＳＣＨはビット操作命令と同じ分類にしているが、かなり異なった性質をもっている。ＢＳＣＨ命令でも、他のビット操作命令と同じようにオフセットを自由に設定できる方が使いやすいという考え方があるが、その目的でＢＶＳＣＨ命令が別に設けられているため、ＢＳＣＨとしてはできるだけ軽い仕様に絞り、オフセットの範囲を制限している。オフセットの有効範囲は、他のビット操作命令でレジスタ直接モードＲｎを指定した場合と同じ範囲である。また、一般のビット操作命令ではｏｆｆｓｅｔがｒｅａｄ－ｏｎｌｙ、ｂａｓｅがｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅとなっているのに対して、ＢＳＣＨではそれとは逆にｏｆｆｓｅｔがｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅ、ｄａｔａ（ｂａｓｅ　ａｄｄｒｅｓｓ）がｒｅａｄ－ｏｎｌｙとなっており、注意が必要である。ＢＳＣＨ／Ｆで、指定したビットが見付からなかった場合には、サーチを行なった最後のビット（ワード境界）の次のビットのオフセットがセットされ、Ｖ＿ｆｌａｇ＝１となる。サーチ失敗時にも、ＥＩＴは起動しない。結果的に、サーチを行なったビット数の分だけオフセットが加算される。

117

[例]

@mem1 = H'00000000,R0 = 0,big-endianで
BSCH/0/F　　@mem1.W,R0を実行
==>R0=0のまま、 V__flagは0

@mem1 = H'ffff7fff,R0 = 0,big-endianで
BSCH/0/F　　@mem1.W,R0を実行
==>R0=16 となり、 V__flagは0

@mem1 = H'ffffffff,R0 = 0,big-endianで
BSCH/0/F　　@mem1.W,R0を実行
==>R0=32 となり、 V__flagは1

ＢＳＣＨ／Ｂで、指定したビットが見付からなかった場合には、ｏｆｆｓｅｔに（－１）がセットされる。この＊

118

＊場合もＶ__ｆｌａｇがセットされるが、ＥＩＴは起動されない。ＢＳＣＨ命令では、ｏｆｆｓｅｔの初期値の上位ビットは無視されるが、命令終了後にセットされるｏｆｆｓｅｔ値（サーチ結果）については、上位ビットまで意味を持つ値になっている。つまり、ｏｆｆｓｅｔの上位ビットに、ワード境界を越えた分のビットオフセットや実効アドレスなどといった別の情報が入っていても、ＢＳＣＨ命令実行後は、ｏｆｆｓｅｔの上位ビットも書き換えられてしまうわけである。サーチ成功の時はｏｆｆｓｅｔが０～３１の値をとる（ｄａｔａが３２ビットの場合）ので、／Ｆ，／Ｂとも上位ビットは常に０となる。また、／Ｆでサーチ失敗の場合はｏｆｆｓｅｔ＝３２となるため、上位ビットが００．．．．００１に、下位ビットが０００００になる。／Ｂでサーチ失敗の場合はｏｆｆｓｅｔ＝（－１）となるため、上位ビットが１１．．．．１１１に、下位ビットが１１１１１になる。

[例]

@mem1 = H'00000000,R0 = H'00000020 で
BSCH/0/F　　@mem1.W,R0.Wを実行
==>R0 = H'00000000 となる。
（R0 = H'00000020 のままではない）

@mem1 = H'ffff7fff,R0 = H'00000020 で
BSCH/0/F　　@mem1.W,R0.Wを実行
==>R0 = H'00000010 となる。
（R0 = H'00000030 ではない）

@mem1 = H'ffffffff,R0 = H'12345678 で
BSCH/0/F　　@mem1.W,R0.Wを実行
==>サーチ失敗のため、R0 = H'00000020, V__flag=1となる。

@mem1 = H'ffffffff,R0 = H'00000020 で
BSCH/0/F.　　@mem1.W,R0.Wを実行
==>サーチ失敗のため、 V__flag=1となる。
R0はH'00000020のまま。
R0 = H'00000040 （上位に桁上げ伝播）ではない。

※【０１６６】［プログラム例外］

※40

・予約命令例外
・RR='11' のとき
・MM='11' のとき
・EaR が@-SPのとき
・EaM が#imm__data,@SP+,@-SPのとき

【０１６７】１２－７．固定長ビットフィールド操作命令

「本発明装置」の一つのデータタイプであるビットフィールドは、ビットフィールド中のＭＳＢの位置、およびビットフィールドの長さ（ｗｉｄｔｈ）により指定される。ビットフィールドのＭＳＢの位置は、ｂａｓｅとｏｆｆｓｅｔとの組で示される。ｂａｓｅで示されるメモリのＭＳＢ（第０ｂｉｔ）がｏｆｆｓｅｔ＝０になる。

ｏｆｆｓｅｔの意味は、ビット操作命令の時と同じである。ビットフィールドとｂａｓｅ、ｏｆｆｓｅｔ、ｗｉｄｔｈとの関係を図１４７に示す。

［メモリ上のビットフィールド操作をする場合］図１４７に示すように太線内が対象ビットフィールドである。固定長ビットフィールドの操作命令（ＢＦＥＸＴ，ＢＦＥＸＴＵ，ＢＦＣＭＰ，ＢＦＣＭＰＵ，ＢＦＩＮＳ，ＢＦＩＮＳＵ）は、ＡＩ向き応用のタグ処理（タグの比較やタグの切り出し）などに特に有効である。固定長ビットフィールド命令には、次の２つのフォーマットがある。

・ｏｆｆｓｅｔを８ビットの一般形アドレッシングモードで指定し、ｗｉｄｔｈをレジスタで指定する形式。これを’：Ｇ’フォーマットと呼ぶ。’：Ｇ’フォーマットでは、ｂａｓｅにｏｆｆｓｅｔ／８の値を加えることにより、実際にアクセスするメモリのアドレスが決まる。２６ｂｉｔ以上のビットフィールドで５バイトにまたがるビットフィールドを扱うこともできる。

・ｏｆｆｓｅｔを８ビットのイミディエート値で指定し、ｗｉｄｔｈをリテラルで指定する形式。これを’：Ｅ’フォーマットと呼ぶ。’：Ｅ’フォーマットでは、ワード境界を越えないビットフィールドのみを対象として速度を上げるために、ｂａｓｅの１ワードからはみ出した部分のビットフィールドについて、動作を保証しないものとしている。ｗｉｄｔｈ＋ｏｆｆｓｅｔ≧ｓｉｚｅであってもＥＩＴは起動しないが、読みだし時、書き込み時とも値が不定となる。ｂａｓｅの１ワードのみのアクセスでも命令仕様を実現することができるので、ｏｆｆｓｅｔとは無関係に、ｂａｓｅのみを見て操作対象となるビットフィールドのメモリアドレスを決めることが可能である。したがって、インプリメント次第で命令実行を高速化することもできる。ＢＦ：ＥとＢＦ：Ｇのｂａｓｅで許されるアドレッシングモードは、全く同じである。ＢＦＩＮＳ，ＢＦＩＮＳＵ，ＢＦＣＭＰ，ＢＦＣＭＰＵでは、：Ｇ，：Ｅフォーマットのそれぞれに対して、さらに次の二つの形式がある。

・ｓｒｃオペランドをレジスタで指定する：Ｒフォーマット

・ｓｒｃオペランドをイミディエートで指定する：Ｉフォーマット

ｗｉｄｔｈの値は、１～３２（〈〈ＬＸ〉〉では１～６４）に制限されており、命令実行前に０＜ｗｉｄｔｈ≦

３２（６４）のチェックが行なわれる。ｗｉｄｔｈ＝０の場合もエラーである。違反した場合には、不正オペランド例外（ＩＯＥ）となる。すべての命令について、ｏｆｆｓｅｔ，ｗｉｄｔｈとも符号付きの数として扱われる。ただし、ｗｉｄｔｈはもともと１～３２（６４）の値しか許されていないため、符号付きと考えるか符号なしと考えるかは実際の動作には影響せず、仕様書記述上の問題となっている。また、：Ｅフォーマットの命令のｏｆｆｓｅｔも符号付きとして扱われ、この場合はｏｆｆｓｅｔとして－１２８～＋１２７の数を表わすことになる。（ただし、後で述べるように、：Ｅフォーマットではｂａｓｅのアドレスの１ワードｂａｓｅ～ｂａｓｅ＋３からハミ出すビットフィールドについて、ハミ出した部分に関する動作を保証していない。）

ＢＦ命令のビットフィールドでない方のオペランドは、普通の整数として扱われる。したがって、例えばＢＦＥＸＴの場合は、抽出されたビットフィールドがレジスタの　ＬＳＢ側に詰めてセットされ、ＭＳＢ方向に符号拡張される。ビット位置＝０（ＭＳＢ）に合わせてビットフィールドをセットするのではない。ｂａｓｅとしてレジスタを対象とした場合には、ビットフィールドは一つのレジスタの範囲内に限られる。レジスタ対象の固定長ビットフィールド命令も「本発明装置」でサポートする。ただしこれは〈〈Ｌ２〉〉となっている。これは、レジスタを対象としたビットフィールド操作の場合、現段階では、ＢＦ：Ｅ命令よりもシフト命令とＡＮＤ命令を組み合せて実行する方が速くなる可能性があるためである。レジスタ対象のビットフィールド命令（〈〈Ｌ２〉〉）については、：Ｇでも次に述べる：Ｅと同じように、１ワード（レジスタ）からハミ出すビットフィールドについて、ハミ出した部分に関する動作を保証しないものとする。ＢＦＥＸＴ，ＢＦＥＸＴＵでは不定値が得られ、ＢＦＩＮＳ，ＢＦＩＮＳＵでは無視される。ｏｆｆｓｅｔ＋ｗｉｄｔｈ≧ｓｉｚｅの場合もＥＩＴは起動しない。：Ｅフォーマットでは、対象となるビットフィールドのうちで、ｓｉｚｅを越えたビットオフセットを持つ部分についてのみ動作を保証していない。同様に、負のビットオフセットを持つ部分についても動作を保証していない。いずれの場合も、指定されたビットフィールドのうちで、ｂａｓｅのアドレスで示される１ワードに含まれる部分については、正しく実行される。

121　　122

[例]

address　　N-1　　N

data　　B'abcdefgh　　B'ijklmnop

N+1

B'qrstuvwx

(a～x: 0または1)

の場合、

BFEXT:E.W #3.#9.@N.R0

==>　　R0 = B'lmnopqrst　となる。

BFEXT:E.W #-5.#9.@N.R0

==>　　R0 = B'?????ijkl　となる。(? は不定値)

ｗｉｄｔｈ，ｓｒｃ，ｄｅｓｔのレジスタのサイズ指定は、Ｘフィールドによって共通に行なわれる。サイズ指定フィールドＸは、３２ビット演算と６４ビット演算（〈〈ＬＸ〉〉）の切り換えを行なうものであるが、具体的には次の３つの意味を持つ。

①ｓｒｃ（ｄｅｓｔ）レジスタのサイズ指定（：Ｒフォーマット）

②ｗｉｄｔｈレジスタのサイズ指定（：Ｇフォーマット）

③ｗｉｄｔｈの範囲の指定

X=0 の時　0 <width ≦32

X=1 の時　0 <width ≦64

：Ｅ：Ｉフォーマットの場合①②は意味を持たないが、③の区別を行なうためにやはりＸフィールドを使用する。すなわち、Ｘフィールドは、３２ビットと６４ビットの互換性を高めるためのビットであると言える。：Ｉフォーマットの命令でＳＳ≠００の時には、＃ｉＳ８のフィールドは使用しない。この時、もし＃ｉＳ８のフィールドが０になっていなくても、単に無視される。ただし、マニュアル上は、＃ｉＳ８のフィールドには０を入れるようにしておく。ビットフィールド命令のフォーマットと、それに対して使用できるサイズを列挙すると、図１４８，図１４９のようになる。ビットフィールド命令についても、ビット操作命令と同じようにアクセスを行なうメモリの範囲が問題となるが、インプリメントへの依存性が強いので、強い規定は設けない。［これについては詳細仕様調整中］

〔ニモニック〕

ＢＦＥＸＴ　ｏｆｆｓｅｔ，ｗｉｄｔｈ，ｂａｓｅ，ｄｅｓｔ

〔命令の機能〕

ｅｘｔｒａｃｔ　ｂｉｔ　ｆｉｅｌｄ（ｓｉｇｎｅｄ）ビットフィールドの抽出（符号付き）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１５０に示す。

〔フラッグ変化〕図１５１に示す。

〔解説〕ビットフィールドの抽出を行ない結果をデスティネーションに転送する。ビットフィールドのｗｉｄｔｈよりもデスティネーションのサイズの方が大きい時は、データが符号拡張される。また、ＢＦＥＸＴ：Ｇのｏｆｆｓｅｔも符号拡張される。ＥａＲｂｆのアドレッシングモードでは、@－ＳＰ，@ＳＰ＋，＃ｉｍｍ＿ｄａｔａのモードは使用できない。ｂａｓｅのレジスタ直接モードＲｎは〈〈Ｌ２〉〉であるが「本発明装置」ではサポートする。

［オペレーション］

123　　124

destの初期値を　　[D0.D1....Dd-2.Dd-1]

d=32.64

destに設定される値を[R0.R1....Rd-2.Rd-1]

d=32.64

offset = o.width = w

とする。offset,widthは符号付きとして扱われる。（width ≦ 0.width> dの時はエラー）

この時、抽出される部分のビットフィールドとフラッグの変化は、次のようになる。

（d ≧w の時）

baseのbit0

↓

[...B0.B1....Bo-2.Bo-1.Bo.Bo+1....Bo+w-2.Bo+w-1.Bo+w.Bo+w+1...]

この部分を符号拡張してdestに設定する

[　Bo.　Bo+1....Bo+w-2.Bo+w-1] ==>

[Bo.Bo.....Bo.　Bo.　Bo+1....Bo+w-2.Bo+w-1] ==>

d-w ビットだけ符号拡張される

[R0.R1...Rd-w-1.Rd-w.Rd-w+1.....Rd-2.Rd-1]　（destに設定される）

（d＜wの時）　　「本発明装置」３２では起り得ないケースである。

baseのbit0

↓

[...B0.B1.....Bo-1.Bo.Bo+1,...Bo+w-d-1.Bo+w-d....Bo+w-2.Bo+w-1.Bo+w...]

この部分はカットされる　この部分をdestに設定する

[Bo.Bo+1....Bo+w-d-1.Bo+w-d....Bo+w-2.Bo+w-1] ==>

[Bo+w-d....Bo+w-2.Bo+w-1] ==>

この部分がカットされる

[　R0.....Rd-2.　Rd-1]　（destに設定される）

M_flag　R0

または（d ≧w の時）　Bo

（d ＜w の時）　Bo+w-d

Z_flag　[R0 ～d-1] = 0

または（d ≧w の時）　[Bo ～o+w-1]= 0

（d ＜w の時）　[Bo+w-d ～o+w-1] = 0

V_flag☆　S[Bo ～o+w-1]< - 2^(d-1) .or.

125　　126

S[Bo ～o+w-1]≧ + 2^(d-1)

または（d ≧w の時）　0

（d ＜w の時）　Bo=Bo+1=...=Bo+w-d-1=Bo+w-d の時クリア、それ以外の場合にセットとなる。

「本発明装置」３２であれば常にクリアされる。

【０１６８】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・+='0' のとき

・X ='1' のとき

・EaR が@-SPのとき

・EaRbf が#imm__data,@SP+,@-SPのとき

・不正オペランド例外

・width ≦0,width＞32のとき

〔ニモニック〕

ＢＦＥＸＴＵ　ｏｆｆｓｅｔ，ｗｉｄｔｈ，ｂａｓｅ，ｄｅｓｔ

〔命令の機能〕

ｅｘｔｒａｃｔ　ｂｉｔ　ｆｉｅｌｄ（ｕｎｓｉｇｎｅｄ）

ビットフィールドの抽出（符号なし）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１５２に示す。

〔フラッグ変化〕図１５３に示す。

〔解説〕ビットフィールドの抽出を行ない結果をデスティネーションに転送する。ビットフィールドのｗｉｄｔｈよりもデスティネーションのサイズの方が大きい時は、データがゼロ拡張される。ただし、ＢＦＥＸＴＵ：Ｇのｏｆｆｓｅｔは符号拡張される。ＥａＲｂｆのアドレッシングモードでは、@－ＳＰ，@ＳＰ＋，＃ｉｍｍ＊

*__ｄａｔａのモードは使用できない。ｂａｓｅのレジスタ直接モードＲｎは〈〈Ｌ２〉〉であるが「本発明装置」ではサポートする。

［オペレーション］

destの初期値を　[D0.D1....Dd-2.Dd-1]

d=32,64

destに設定される値を[R0.R1....Rd-2.Rd-1]

d=32.64

offset = o,width = w

とする。ｏｆｆｓｅｔ，ｗｉｄｔｈは符号付きとして扱われる。（ｗｉｄｔｈ≦０，ｗｉｄｔｈ＞ｄの時はエラー）

この時、抽出される部分のビットフィールドとフラッグの変化は、次のようになる。

（ｄ≧ｗの時）

baseのbit0

↓

[...B0.B1....Bo-2.Bo-1.Bo.Bo+1....Bo+w-2.Bo+w-1.Bo+w.Bo+w+1...]

この部分をゼロ拡張してｄｅｓｔに設定する

[　Bo.　Bo+1....Bo+w-2.Bo+w-1] ==>

[0. 0.....0.　Bo.　Bo+1....Bo+w-2.Bo+w-1] ==>

d-wビットだけゼロ拡張される

[R0.R1...Rd-w-1.Rd-w.Rd-w+1.....Rd-2.Rd-1]（destに設定される）

（ｄ＜ｗの時）

「本発明装置」３２では起り得ないケースである。

127　　128

baseのbit0

↓

[...B0. B1.....Bo-1. Bo. Bo+1....Bo+w-d-1.

この部分はカットされる

Bo+w-d....Bo+w-2. Bo+w-1. Bo+w...]

この部分をdestに設定する

[Bo. Bo+1....Bo+w-d-1. Bo+w-d....Bo+w-2. Bo+w-1] ==>

[Bo+w-d....Bo+w-2. Bo+w-1] ==>

この部分がカットされる

[　R0.....Rd-2.　Rd-1]

(destに設定される)

M__flag　　R0

または (d ＞w の時)　0

(d ＝w の時)　Bo

(d ＜w の時)　Bo+w-d

Z__flag　　[R0 ～d-1] = 0

または (d ≧w の時)　[Bo ～o+w-1] = 0

(d ＜w の時)　[Bo+w-d ～o+w-1] = 0

V__flag☆　　U[Bo～o+w-1]≧ + 2^d

または (d ≧w の時)　0

(d ＜w の時)　Bo=Bo+1= ...=Bo+w-d-1=0 の時クリア、それ以外の場合にセットとなる。

「本発明装置」３２であれば常にクリアされる。

【０１６９】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・+='0' のとき

・X ='1' のとき

・EaR が@-SPのとき

・EaRbf が#imm__data,@SP+,@-SPのとき

・不正オペランド例外

・width ≦ 0, width＞32のとき

〔ニモニック〕

ＢＦＩＮＳ　ｓｒｃ，ｏｆｆｓｅｔ，ｗｉｄｔｈ，ｂａｓｅ

〔命令の機能〕

ｉｎｓｅｒｔ　ｂｉｔ　ｆｉｅｌｄ

ビットフィールドの挿入（符号付き）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１５４に示す。

〔フラッグ変化〕図１５５に示す。

〔解説〕ソースの値をビットフィールドに挿入する。ソースのサイズよりもビットフィールドのｗｉｄｔｈの方が大きい時は、データが符号拡張される。また、ＢＦＩＮＳ：Ｇのｏｆｆｓｅｔも符号拡張される。ＥａＲｂｆのアドレッシングモードでは、@－ＳＰ，@ＳＰ＋，#ｉｍｍ__ｄａｔａのモードは使用できない。ｂａｓｅのレジスタ直接モードＲｎは〈（Ｌ２）〉であるが「本発明装置」ではサポートする。

〔オペレーション〕

srcの初期値を　[S0. S1....Ss-2. Ss-1]

s=8, 16, 32, 64(:I)

s=32, 64(:R)

offset = o, width = w

とする。ｏｆｆｓｅｔ，ｗｉｄｔｈとも符号付きとして扱われる。（ｗｉｄｔｈ≦０，ｗｉｄｔｈ＞ｄの時はエ

ラー)

この時、挿入される部分のビットフィールドとフラグの変化は、次のようになる。

baseのbit0

↓

[...B0.B1....Bo-1.Bo.Bo+1....Bo+w-s-1.Bo+w-s.Bo+w-s+1....Bo+w-1.Bo+w

...]] ==>

[...B0.B1....Bo-1.S0 .S0.....S0. S0. S1.....Ss-1.Bo+w

src がw-s ビットだけ符号拡張される...]]

＊（ｗ≧ｓの時)　ビットフィールドの変化

＊

（ｗ＜ｓの時)　ビットフィールドの変化

baseのbit0

↓

[...B0.B1....Bo-2.Bo-1. Bo. Bo+1.... Bo+w-1.Bo+w...]] ==>

[...B0.B1....Bo-2.Bo-1.Ss-w.Ss-w+1.....Ss-1.Bo+w...]]

src の[S0.S1....Ss-w-1] がカットされる。

M__flag　対象ビットフィールドのMSB(Bo) の変化を基準とする。
または（w ≧s の時）　S0
（w ＜s の時）　Ss-w

Z__flag　対象ビットフィールド[Bo ～o+w-1]の変化を基準とする。
または（w ≧s の時）　[S0 ～ s-1] = src = 0
（w ＜s の時）　[Ss- w～s-1] = 0

V__flag☆　S[S0～s-1]= src＜ - 2^(w-1) .or.
S[S0～s-1]= src≧ + 2^(w-1)

または（w ≧s の時）　0
（w ＜s の時）　S0=S1=...=Ss-w-1=Ss-w の時クリア、それ以外の場合にセットとなる。

【０１７０】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・+='0' のとき

・X ='1' のとき

・SS='11' のとき

・EaR が@-SPのとき

・EaMbf が#imm__data.@SP+,@-SPのとき

・不正オペランド例外

・width ≦0,width>32のとき

〔ニモニック〕

ＢＦＩＮＳＵ　ｓｒｃ，ｏｆｆｓｅｔ，ｗｉｄｔｈ，ｂａｓｅ

〔命令の機能〕

insert bit field

ビットフィールドの挿入（符号なし）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１５６に示す。

〔フラッグ変化〕図１５７に示す。

〔解説〕ソースの値をビットフィールドに挿入する。ソースのサイズよりもビットフィールドのｗｉｄｔｈの方が大きい時は、データがゼロ拡張される。一方、ＢＦＩＮＳＵ：Ｇのｏｆｆｓｅｔは符号拡張される。ＥａＲｂｆのアドレッシングモードでは、@－ＳＰ，@ＳＰ＋，

131

＃ｉｍｍ＿ｄａｔａのモードは使用できない。ｂａｓｅのレジスタ直接モードＲｎは〈〈Ｌ２〉〉であるが「本発明装置」ではサポートする。

［オペレーション］

srcの初期値を　[S0.S1....Ss-2.Ss-1]

s=8,16,32,64(:I)

s=32,64(:R)

offset = o,width = w

＊10

baseのbit0

↓

[...B0.B1....Bo-1.Bo.Bo+1....Bo+w-s-1.Bo+w-s.Bo+w-s+1....Bo+w-1.Bo+w

...]] ==>

[...B0.B1....Bo-1.0.　0.....0.　S0.　S1.....Ss-1.Bo+w

srcがw-s ビットだけ符号拡張される

...]]

（ｗ＜ｓの時）

baseのbit0

↓

[...B0.B1....Bo-2.Bo-1.　Bo.　Bo+1....Bo+w-1.Bo+w...]] ==>

[...B0.B1....Bo-2.Bo-1.Ss-w.Ss-w+1.....Ss-1.Bo+w...]]

src の[S0.S1....Ss-w-1] がカットされる。

M＿flag　対象ビットフィールドのMSB(Bo) の変化を基準とする。

または（w ＞s の時） 0

（w ＝s の時） S0

（w ＜s の時） Ss-w

Z＿flag　対象ビットフィールド[Bo ～ o+w-1] の変化を基準とする。

または（w ≧s の時） [S0 ～s-1] = src = 0

（w ＜s の時） [Ss-w ～s- = 0

V＿flag☆　U[S0 ～s-1]＝ src≧ + 2^w

または（w ≧s の時） 0

（w ＜s の時） S0=S1=...=Ss-w-1=0 の時クリア、それ以外の場合にセットとなる。

132

＊とする。ｏｆｆｓｅｔ，ｗｉｄｔｈとも符号付きとして扱われる。（ｗｉｄｔｈ≦０，ｗｉｄｔｈ＞ｄの時はエラー）

この時、挿入される部分のビットフィールドとフラッグの変化は、次のようになる。

（ｗ≧ｓの時）

ビットフィールドの変化

ビットフィールドの変化

【０１７１】〔プログラム例外〕

133

・予約命令例外

・RR='11' のとき

・+='0' のとき

・X ='1' のとき

・SS='11' のとき

・EaR が@-SPのとき

・EaMbf が#imm_data,@SP+,@-SPのとき

・不正オペランド例外

・width ≦0,width>32のとき

〔ニモニック〕

ＢＦＣＭＰ　ｓｒｃ，ｏｆｆｓｅｔ，ｗｉｄｔｈ，ｂａｓｅ

〔命令の機能〕

ｃｏｍｐａｒｅ　ｂｉｔ　ｆｉｅｌｄ（ｓｉｇｎｅｄ）ビットフィールドの比較（符号付き）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１５８に示す。

〔フラッグ変化〕図１５９に示す。

〔解説〕ソースｓｒｃの値とビットフィールドの値を比較する。ソースのサイズとビットフィールドのｗｉｄｔｈの値が異なっている場合は、サイズの小さい方のデータが符号拡張されてから比較される。また、ＢＦＣＭＰ：Ｇのｏｆｆｓｅｔも符号拡張される。ＥａＲｂｆのアドレッシングモードでは、@－ＳＰ，@ＳＰ＋，＃ｉｍｍ＿ｄａｔａのモードは使用できない。ｂａｓｅのレジスタ直接モードＲｎは〈〈Ｌ２〉〉であるが「本発明装置」ではサポートする。

134

［オペレーション］

srcの初期値を　[S0.S1....Ss-2.Ss-1]

s=8,16,32,64(:I)

s=32,64(:R)

offset = o.width = w

とする。ｏｆｆｓｅｔ，ｗｉｄｔｈとも符号付きとして扱われる。（ｗｉｄｔｈ≦０，ｗｉｄｔｈ＞ｄの時はエラー）

この時、比較される部分のビットフィールドとフラッグの変化は、次のようになる。

（ｓ≧ｗの時）

baseのbit0

↓

[...B0.B1....Bo-2.Bo-1.Bo.Bo+1....Bo+w-2.Bo+w-1.Bo+w.Bo+w+1...]

この部分を符号拡張してsrc と比較する

（ｓ＜ｗの時）

baseのbit0

↓

[...B0.B1.....Bo-1.Bo.Bo+1....Bo+w-s-1.Bo+w-s....Bo+w-2.Bo+w-1.Bo+w.

..]

ｓｒｃを符号拡張してこの部分と比較する

L_flag　S[Bo ～o+w-1] - S[S0 ～s-1]＜ 0

比較結果によってセットされる。

Z_flag　S[Bo ～o+w-1] - S[S0 ～s-1]＝ 0

比較結果によってセットされる。

・予約命令例外

・RR='11' のとき

・+='0' のとき

・－ ='1' のとき

・SS='11' のとき

・EaR が@-SPのとき

・EaRbf が#imm_data,@SP+,@-SPのとき

・不正オペランド例外

・width ≦0,width>32のとき

〔ニモニック〕

【０１７２】〔プログラム例外〕

135

ＢＦＣＭＰＵ　ｓｒｃ，ｏｆｆｓｅｔ，ｗｉｄｔｈ，ｂａｓｅ

〔命令の機能〕

ｃｏｍｐａｒｅ　ｂｉｔ　ｆｉｅｌｄ（ｕｎｓｉｇｎｅｄ）

ビットフィールドの比較（符号なし）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１６０に示す。

〔フラッグ変化〕図１６１に示す。

136

＊〔解説〕ソースｓｒｃの値とビットフィールドの値を比較する。ソースのサイズとビットフィールドのｗｉｄｔｈの値が異なっている場合は、サイズの小さい方のデータがゼロ拡張されてから比較される。一方、ＢＦＣＭＰＵ：Ｇのｏｆｆｓｅｔは符号拡張される。ＥａＲｂｆのアドレッシングモードでは、@－ＳＰ，@ＳＰ＋，＃ｉｍｍ＿ｄａｔａのモードは使用できない。ｂａｓｅのレジスタ直接モードＲｎは〈〈Ｌ２〉〉であるが「本発明装置」ではサポートする。

＊10　〔オペレーション〕

srcの初期値を　[S0.S1....Ss-2.Ss-1]

s=8,16,32,64(:I)

s=32,64(:R)

offset = o.width = w

とする。ｏｆｆｓｅｔ，ｗｌｄｔｈとも符号付きとして扱われる。（ｗｌｄｔｈ≦０，ｗｉｄｔｈ＞ｄの時はエラー）

※この時、比較される部分のビットフィールドとフラッグの変化は、次のようになる。

※　（ｓ≧ｗの時）

baseのbit0

↓

[...B0.B1....Bo-2.Bo-1.Bo.Bo+1....Bo+w-2.Bo+w-1.Bo+w.Bo+w+1...]

この部分をゼロ拡張してsrc と比較する

（ｓ＜ｗの時）

baseのbit0

↓

[...B0.B1.....Bo-1.Bo.Bo+1....Bo+w-s-1.Bo+w-s....Bo+w-2.Bo+w-1.Bo+w.

..]

ｓｒｃをゼロ拡張してこの部分と比較する

L_flag　U[Bo ～o+w-1] - U[S0 ～s-1] < 0

比較結果によってセットされる。

Z_flag　U[Bo ～o+w-1] - U[S0 ～s-1] = 0

比較結果によってセットされる。

【０１７３】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・+='0' のとき

・- ='1' のとき

・SS='11' のとき

・EaR が@-SPのとき

・EaRbf が#imm_data,@SP+,@-SPのとき

・不正オペランド例外

・width ≦０,width＞32のとき

【０１７４】１２－８．任意長ビットフィールド操作命令

任意長のビットフィールド操作命令には、次のようなものがある。

一般的な演算と転送	BVMAP
転送	BVCPY
繰り返しパターンの演算と転送	BVPAT
０または１のサーチ	BVSCH

40

ＢＶＭＡＰ，ＢＶＰＡＴ，ＢＶＣＰＹは、ビットマップディスプレイ上のウインドウ操作（ｂｉｔｂｌｔ）を主な目的とした命令である。説明のため、ビットマップディスプレイの属性について用語を定義する。

（ｃｏｌｏｒ　ｓｃａｌｅ，ｃｏｌｏｒ　ｏｆｆｓｅｔとｂｉｔ－ｄｏｔ極性）

・ｃｏｌｏｒ　ｓｃａｌｅ　：１ｄｏｔを連続の何ｂｉｔで表すか。

50

137　　138

例：

<color scale = 1>

1bit が1 dot。1バイトで連続した8 dot。

白黒のビットマップディスプレイ。

または、color を構成する各ビットをバンクにしたビットマップディスプレイ。

<color scale = 4>

隣合った4bit で1 dot。1バイトで連続した2 dot。

16色カラーのビットマップディスプレイ。

・ｂｉｔ－ｄｏｔ極性

これは、ビットマップディスプレイとプロセッサとの組み合わせに対して生ずる概念である。小さいアドレスの方が左側に表示されるような一般的なビットマップディスプレイにおいて、小さいビット番号に対応するドットも左側に表示される場合、このようなビットマップディスプレイを正のｂｉｔ－ｄｏｔ極性を持つという。また、逆のものを負のｂｉｔ－ｄｏｔ極性を持つという。つまり、ｂｉｇ－ｅｎｄｉａｎのプロセッサでは、ＭＳＢビットが左側に表示される場合に正のｂｉｔ－ｄｏｔ極性を持つことになる。

・ｃｏｌｏｒ　ｏｆｆｓｅｔ　　＊

```
bit offset = X * color scale + color offset
```

負のｂｉｔ－ｄｏｔ極性の時

```
bit offset = (X * color scale + color offset) .xor. 7
```

＊１ｄｏｔを構成する複数のビットのうち、第何ビットを操作するか、ということ。

```
0 ≦ color offset < color scale
```

という関係が成り立つ。これは、ビットマップディスプレイハードウエアの属性ではなく、ビットマップディスプレイ操作上のパラメータである。ｂａｓｅ　ａｄｄｒｅｓｓに対応するドットから、横方向にＸ（ｄｏｔ　ｏｆｆｓｅｔ）だけ変位したドットを操作する場合、そのメモリ上のｂｉｔ　ｏｆｆｓｅｔは次のように計算される。（ｄｏｔ　ｏｆｆｓｅｔは画面上の点の概念、ｂｉｔ　ｏｆｆｓｅｔはメモリ上のｂｉｔの概念である。）

正のｂｉｔ－ｄｏｔ極性の時

ところで、実際の「本発明装置」のＢＶＭＡＰ，ＢＶＣＰＹ，ＢＶＰＡＴ命令では、インプリメントのことを考えて制約を設け、次のような場合にのみ使用できるようになっている。

・ｂｉｔ－ｄｏｔ極性が正

・ｃｏｌｏｒ　ｓｃａｌｅが１

このため、ビットマップディスプレイのハードウエアをある程度規定することはやむを得ない。具体的な制限は次のようになる。

・ｂｉｔ－ｄｏｔ極性が正なので、「本発明装置」をｂｉｇ－ｅｎｄｉａｎとした場合には、アドレスの小さい方、ビット番号の小さい方（ＭＳＢ）が画面の左側に表示されなければならない。

・ｃｏｌｏｒ　ｓｃａｌｅ＝１のみなので、ｃｏｌｏｒ　ｓｃａｌｅ≠１のビットマップディスプレイに対しては次のような制約がある。ｃｏｌｏｒ　ｓｃａｌｅ≠１のビットマップディスプレイでは、ｃｏｌｏｒ　ｏｆｆｓｅｔ毎に演算の種類を変えることができなくなる。ＢＶＭＡＰ命令でｃｏｌｏｒ　ｓｃａｌｅの変更をすることはできないので、ビットマップディスプレイのｃｏｌｏｒ　ｓｃａｌｅが１でない時には、内部表現も同じｃｏｌｏｒ　ｓｃａｌｅにしないと、ＢＶＭＡＰ命令は、使えない。その場合、画面イメージの内部表現がハードウエアに依存することになるので、他のハードウエアとの間でデータの転送を行なう時は、データ形式の変換が必要になる。任意長ビットフィールド操作命令はオペランドが多く、実行時間も長い。したがって、実行中での割り込み受け付けや、割り込み処理後の再実行のメカニズムが必要である。「本発明装置」では、オペランドの指定と演算の進行状況の表現のために固定番号のレジスタを使用している。そのため、任意長ビットフィールド命令実行中に割り込みが入っても、割り込み処理ハンドラ中でそのレジスタの退避と復帰が正しく行なわれていれば、割り込み処理後に、そのビットフィールド命令を途中から再開できる。実行中断後に状態の退避やコンテキストスイッチを行なったり、コンテキストスイッチ後に別のプロセスで同じビットマップ命令を実行し、再び前のコンテキストに戻って前のビットマップ命令を再開したとしても、問題なく動かなければならない。また、ＢＴＲＯＮの仕様では、ＶＲＡＭではない普通のメインメモリに対しても、文字や図形の描画が行なわれることがある。したがって、任意長ビットフィールド命令でもページフォールトの起こる可能性があり、ストリング命令と同様に、ページフォールトによる実行中断にも対処できなければならない。ＢＶＭＡＰ，ＢＶＣＰＹ命令では、インサートエディタなどで真横に図形を移動することを考え、ビットマップのソースとデスティネーションのオーバーラップにも対応できるようになっている。具

体的には、ストリング命令と同じように、演算を行なう方向を命令中のオプション／Ｆ，／Ｂとして指定する。ソフトウエアにより適当な方向を判定し、デスティネーションがソースを破壊しないように演算を進める。ただし、インプリメントの負担を考え、逆方向の処理を指定するオプション／Ｂは〈〈Ｌ２〉〉となっている。「本発明装置」ではＢＴＲＯＮの動作を高速化するための逆方向処理もサポートする。ｓｒｃとｄｅｓｔがオーバーラップしていた場合、ｓｒｃのｂａｓｅ～ｏｆｆｓｅｔよりもｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が小さければ、ｏｆｆｓｅｔの小さい方から処理することによってｄｅｓｔがｓｒｃを破壊することなく処理を進めることができる。この目的で／Ｆオプションを使用する。画面とビットマップとの対応は、通常オフセット（アドレス）の小さい方が左側になる。したがって、ｓｒｃよりもｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が小さくなるのは、文字の削除などによってビットマップデータを左に動かそうとした時である。また、ｓｒｃのｂａｓｅ～ｏｆｆｓｅｔよりもｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が大きければ、ｏｆｆｓｅｔの大きい方から処理することによってｄｅｓｔがｓｒｃを破壊することなく処理を進めることができる。この目的で／Ｂオプションを使用する。ｓｒｃよりもｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が大きくなるのは、文字の挿入などによってビットマップデータを右に動かそうとした時である。ｓｒｃとｄｅｓｔがオーバーラップする可能性がある場合には、ソフトウエアの判断により正しいオプションを使用し、ｄｅｓｔがｓｒｃを破壊しないように演算を進める必要がある。ただし、／Ｂオプションは〈〈Ｌ２〉〉となっているので、／Ｂが使用できない場合には、ｓｒｃを一旦他の場所にコピーしてからｄｅｓｔとの演算を行なわなければならない。オーバーラップがない場合には、どちらのオプションを使っても結果は変わらない。ここで、問題は、ｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が小さいのに／Ｂオプションを使用した場合や、ｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が大きいのに／Ｆオプションを使用した場合にどのような動作を行なうかということである。基本的には、既に演算の終った部分のｄｅｓｔが、ｓｒｃのまだ参照されていない部分を破壊する形になるため、正しい結果が得られない。しかも、この時は、アルゴリズム上、命令が途中で中断して再実行を行なった場合に、結果が変わってくることがある。もともと正しい結果を保証していないのであるから、実行中断によって結果が変わったとしても構わないはずであるが、実行中断のなかった場合は正しい結果が得られることもあるので、再現不可能なバグが入りやすい状況になる。しかし、このエラーチェックをきちんと行なう

とオーバーヘッドが増え、実行時間の低下をもたらすので、エラーチェックは行なわない。ユーザの側で注意が必要である。任意長ビットフィールド命令では、レジスタ上のビットオフセットｏｆｆｓｅｔ、ビット幅ｗｉｄｔｈ、パターンデータｐａｔｔｅｒｎのサイズとして、３２ビットまたは６４ビット〈〈ＬＸ〉〉のみが使用できる。８，１６ビットの指定は行なわない。３２ビットと６４ビットのレジスタサイズの選択は、Ｘフィールドによって共通に行なわれる。ＢＶＭＡＰ，ＢＶＣＰＹ，ＢＶＰＡＴ命令のｄｅｓｔ側のメモリアクセス方法については、ｗｒｉｔｅまたはｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅということで特に規定しない。ＢＶ命令でｗｉｄｔｈ≦０の場合には、何もせずに命令を終了し、ＥＩＴとはしない。この時、ＢＶＳＣＨ命令では、長さによる終了を示すＶ＿ｆｌａｇ（サーチ失敗と同じ）がセットされる。これは、ＢＶ命令やストリング命令の様な高機能命令の場合には、その外側でさらに高機能のサブルーチンを作ることが多く、チェックが必要であればそのサブルーチンで行なえばよいと考えているためである。例えば、ＢＶＭＡＰであれば、それをライン数だけ繰り返してＢｉｔＢｌｔ関数を作ることが多く、その時はｗｉｄｔｈがすべて共通になるので、毎回ｗｉｄｔｈをチェックする必要はない。一方、ＢＦ命令のように、コンパイラが直接生成する可能性のあるコードでは、できるだけチェックを厳重にする必要がある。したがって、ＢＦ命令のｗｉｄｔｈは例外で検出するようにしている。任意長ビットフィールド命令でｏｆｆｓｅｔ＋ｗｉｄｔｈがオーバーフローする場合には、割り込みによる命令実行中断時、および命令終了時のレジスタ上のｏｆｆｓｅｔ値がおかしな値になり、正常な命令の実行ができなくなる。この場合は、動作を保証しないものとする。アーキテクチャ上は、命令実行開始時にこれを検出して不正オペランド例外（ＩＯＥ）とするのが望ましいが、チェックのために実行時間が伸びるので、チェックせずに実行するものとする。（なお、ストリング命令の場合は、ｏｆｆｓｅｔに相当するのが整数ではなくポインタアドレスとなっているため、オーバーフローとしては扱わず、単にアドレスがラップアラウンドするだけである。）

〔ニモニック〕

ＢＶＳＣＨ

〔命令の機能〕

ｆｉｎｄ　ｆｉｒｓｔ’０’ｏｒ’１’ｉｎ　ｔｈｅ　ｂｉｔｆｉｅｌｄ（ｖａｒｉａｂｌｅ　ｌｅｎｇｔｈ）

０または１のサーチ（任意長ビットフィールド）

〔命令オプション〕

141　　142

/0　'0' をサーチ（デフォルト）

/1　'1' をサーチ

/F　ビット番号の増加方向にサーチ（デフォルト）

/B　ビット番号の減少方向にサーチ

<<L2>>（「本発明装置」ではサポートする）

〔命令ビットパターンとアセンブラ表記〕図１６２に示す。

〔フラッグ変化〕図１６３に示す。

〔解説〕任意長のビットフィールドの中にある'０'または'１'のビットをサーチする。サーチを開始するビット番号（ｂｉｔ　ｏｆｆｓｅｔ）をｏｆｆｓｅｔオペランド（Ｒ１）にセットしてこの命令を実行すると、命令実行後、サーチ結果のビット番号がｏｆｆｓｅｔオペランド（Ｒ１）にセットされている。つまり、ｏｆｆｓｅｔはｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅとなっている。これは、ビットの検索を繰り返して行なうことを想定したためである。ｏｆｆｓｅｔは符号付き整数として扱われ、その値は任意である。ＢＶＳＣＨを実行した結果、サーチ失敗だった場合には、Ｖ＿ｆｌａｇをセットし、ｏｆｆｓｅｔは次にサーチすべきビットを指示。ＥＩＴは起動しない。ＢＶＳＣＨ命令のオフセットやＶ＿ｆｌａｇの変化方法は、ＢＳＣＨ命令に準じたものである。／Ｂによる逆方向のサーチは〈〈Ｌ２〉〉仕様となっているが「本発明装置」ではサポートする。この命令は、ディスクやメモリの空きブロック検索などに使用することを目的としたものである。なお、任意長ビットフィールド命令やストリング命令などの高機能命令の詳細仕様や、命令終了後のレジスタ値については、付録１１を参照のこと。

【０１７５】〔プログラム例外〕

・予約命令例外

・+='0' のとき

・X='1' のとき

・P='1' のとき

〔ニモニック〕

ＢＶＭＡＰ

〔命令の機能〕

ｂｉｔ　ｏｐｅｒａｔｉｏｎ（ｏｎｅ　ｌｉｎｅ　ＢｉｔＢｌｔ）

ビットマップ演算

〔命令オプション〕

/F　offsetの小さい方から処理する（デフォルト）

/B　offsetの大きい方から処理する

<<L2>>（「本発明装置」ではサポートする）

〔命令ビットパターンとアセンブラ表記〕図１６４に示す。

〔フラッグ変化〕図１６５に示す。

〔解説〕スクリーン上のビットマップ操作を行なうために、任意長のビットフィールドｓｒｃ，ｄｅｓｔに対する各種の論理演算をする命令である。演算の種類はＲ５の下位４ビットで指定され、次の１６種類が用意されている。

【０１７６】

【数１２】

T	True	1 ==> dest
F	False	0 ==> dest
ND	NotDest	~dest ==> dest
D	Dest	dest ==>dest
NS	NotSrc	~src ==> dest
S	Src	src ==>dest
A	And	dest .and. src ==>dest
O	Or	dest .or. src ==>dest
X	Xor	dest .xor. src ==>dest
NA	NotAnd	~dest .and. src ==>dest
NO	NotOr	~dest .or. src ==>dest
AN	AndNot	dest .and. ~src ==>dest
ON	OrNot	dest .or. ~src ==>dest
NAN	NotAndNot	~dest .and. ~src ==>dest
NON	NotOrNot	~dest .or. ~src ==>dest
NX	NotXor	~dest .xor. src ==>dest

【０１７７】このうちＤ（Ｄｅｓｔ）の演算モードは、対称性のために設けられている。ニモニックと実際のビットパターンとの対応については、付録を参照のこと。演算を指定するレジスタＲ５の上位ビットが０でない場合にも、特にチェックは行なわないものとする。ただし、チェックが行なわれていなくても、上位ビットには必ず'０'を入れてもらうように、マニュアル等で指導する必要がある。不正オペランド例外（ＩＯＥ）としないのは、インプリメントの負担が大きく、実行速度に影響が出るためである。／Ｆ，／Ｂオプションは、ｏｆｆｓｅｔの小さい方から処理するか、ｏｆｆｓｅｔの大きい方から処理するかを指定する。これは、ビットマップ

143

のｓｒｃとｄｅｓｔがオーバーラップしていた場合に、処理の方向を明確にしておかないと、ｄｅｓｔがｓｒｃを破壊して正しい結果が得られないからである。ｓｒｃとｄｅｓｔがオーバーラップしていた場合、ｓｒｃのｂａｓｅ～ｏｆｆｓｅｔよりもｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が小さければ、ｏｆｆｓｅｔの小さい方から処理することによってｄｅｓｔがｓｒｃを破壊することなく処理を進めることができる。この目的で／Ｆオプションを使用する。画面とビットマップとの対応は、通常オフセット（アドレス）の小さい方が左側になる。したがって、ｓｒｃよりもｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が小さくなるのは、文字の削除などによってビットマップデータを左に動かそうとした時である。また、ｓｒｃのｂａｓｅ～ｏｆｆｓｅｔよりもｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が大きければ、ｏｆｆｓｅｔの大きい方から処理することによってｄｅｓｔがｓｒｃを破壊することなく処理を進めることができる。この目的で／Ｂオプションを使用する。ｓｒｃよりもｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が大きくなるのは、文字の挿入などによってビットマップデータを右に動かそうとした時である。なお、ｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が小さいのに／Ｂオプションを使用した場合や、ｄｅｓｔのｂａｓｅ～ｏｆｆｓｅｔの方が大きいのに／Ｆオプションを使用した場合には、結果（ｄｅｓｔ）を保証しないものとする。特に、このような場合には、命令実行中に割り込みやページフォールトなどが発生して命令再実行が起こると、結果が変わってくることもある。ｓｒｃとｄｅｓｔがオーバーラップする可能性がある場合には、ソフトウエアの判断により正しいオプションを使用し、ｄｅｓｔがｓｒｃを破壊しないように演算を進める必要がある。ただし、／Ｂオプションは〈〈Ｌ２〉〉となっているので、／Ｂが使用できない場合には、ｓｒｃを一旦他の場所にコピーしてからｄｅｓｔとの演算を行なわなければならない。「本発明装置」では／Ｂオプションはサポートする。オーバーラップがない場合には、どちらのオプションを使っても結果は変わらない。

←base～offsetが小　　base～offsetが大→

［オーバーラップなし］図１６６に示す。

［オーバーラップあり－destのbase～offsetが小］

図１６７に示す。

［オーバーラップあり－destのbase～offsetが大］

図１６８に示す。

【０１７８】〔プログラム例外〕

144

・予約命令例外

・Q='1' のとき

・X='1' のとき

・P='1' のとき

〔ニモニック〕

ＢＶＣＰＹ

〔命令の機能〕

ｂｉｔ　ｔｒａｎｓｆｅｒ

ビットマップ転送

〔命令オプション〕

/F　offsetの小さい方から処理する（デフォルト）

/B　offsetの大きい方から処理する

<<L2>>（「本発明装置」ではサポートする）

〔命令ビットパターンとアセンブラ表記〕図１６９に示す。

〔フラッグ変化〕図１７０に示す。

〔解説〕スクリーン上のビットマップ操作を行なうために、任意長のビットフィールドｓｒｃ，ｄｅｓｔの間の転送をする命令である。この命令は、ＢＶＭＡＰ命令から演算の機能をはずして転送のみに限定し、高速化を目指したものである。／Ｆ，／Ｂオプションの意味はＢＶＭＡＰと同じである。ビットマップのｓｒｃとｄｅｓｔがオーバーラップしていなければ、どちらのオプションを使っても結果は変わらないが、ｓｒｃとｄｅｓｔがオーバーラップした場合には、ソフトウエアの判断により正しいオプションを使用し、ｄｅｓｔがｓｒｃを破壊しないように演算を進める必要がある。／Ｂオプションの場合、Ｒ１，Ｒ４に入れるオフセット値としては、転送の対象となるビットフィールドの最大＋１のオフセット値を指定する。これは、ＳＭＯＶ／Ｂ，ＳＣＭＰ／Ｂの仕様との対応を考えたものである。／Ｂオプションは〈〈Ｌ２〉〉であるが「本発明装置」ではサポートする。

【０１７９】〔プログラム例外〕

・予約命令例外

・Q='1' のとき

・X='1' のとき

・P='1' のとき

〔ニモニック〕

ＢＶＰＡＴ

〔命令の機能〕

ｃｙｃｌｉｃ　ｂｉｔ　ｏｐｅｒａｔｉｏｎ

パターンとビットマップの演算

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１７１に示

145

す。
〔フラッグ変化〕図１７２に示す。
〔解説〕スクリーン上のビットマップをあるパターンで埋めたり、スクリーン上のビットマップとあるパターンの演算を行ないたい場合に使用する命令である。ｐａｔｔｅｒｎを繰り返し発生しながら、ビットフィールドとの論理演算を行なう。演算指定（Ｒ５）の上位ビットが０でない場合は単に無視され、特にチェックは行なわないものとする。ただし、チェックが行なわれていなくても、将来の拡張のため、上位ビットには必ず’０’を入＊れてもらうように、マニュアル等で指導する必要がある。不正オペランド例外（ＩＯＥ）としないのは、インプリメントの負担が大きく、実行速度に影響が出るためである。この命令では、ＢＶＭＡＰ，．ＢＶＣＰＹとは異なり、書き込みの際にシフトは行なわない。ｏｆｆｓｅｔの指定は、単にパターンをクリッピングするだけである。（これに対して、ＢＶＭＡＰ命令では、ｓｒｃとｄｅｓｔのｏｆｆｓｅｔがずれていた場合にはシフトが行なわれる）

146

・予約命令例外
・+='0' のとき
・X='1' のとき
・P='1' のとき

【０１８０】〔プログラム例外〕

【０１８１】１２－９．１０進演算命令
１０進演算に関しては、符号なしＰＡＣＫＥＤ形式（ＢＣＤ）の１０進数の１ワードの加減算とＰＡＣＫ／ＵＮＰＡＣＫ処理をメインプロセッサの〈〈Ｌ１〉〉仕様としてサポートし、符号付きＰＡＣＫＥＤ形式１０進数の１ワードの加減算を〈〈Ｌ２〉〉仕様としてサポートする。また、多桁の１０進数の加減乗除はコプロセッサで行なう。このうち、本章では符号なしＰＡＣＫＥＤ形式１０進数の加減算とＰＡＣＫ／ＵＮＰＡＣＫ処理について説明を行なう。符号付きＰＡＣＫＥＤ形式１０進数をサポートする〈〈Ｌ２〉〉の命令については、後の章で説明を行なう。１０進演算のアドレッシングモードは一般命令と同じになっている。本発明装置では本節で述べる１０進演算命令４種類はサポートしない。
〔ニモニック〕
ＡＤＤＤＸ　ｓｒｃ，ｄｅｓｔ（本発明装置ではサポートしない）
〔命令の機能〕
dest + src + X__flag ==> dest BCD
ＢＣＤの加算
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１７３に示す。
〔フラッグ変化〕図１７４に示す。
〔解説〕パックされたＢＣＤの加算を行なう。８ビット（２桁）、１６ビット（４桁）、３２ビット（８桁）、６４ビット（１６桁）のＢＣＤデータを扱うことができる。ただし、６４ビットは〈〈ＬＸ〉〉仕様である。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースがゼロ拡張された上で加算される。ＢＣＤの数は符号拡張が無意味なので、基本的には符号なしの数と考え、ＡＤＤＤＸのフラッグ変化はＡＤＤＵに準じるものとする。結果がｄｅｓｔに入らない時にＶ＿ｆｌａｇがセットされること、ｄ＜ｓの時はｄｅｓｔのサイズからの桁上げがＸ＿ｆｌａｇにセットされること、などもＡＤＤＵと同様である。ただし、ＡＤＤＵとは異なり、Ｚ＿ｆｌａｇはＡＤＤＸ，ＳＵＢＸのように累積で変化する。ｓｒｃ，ｄｅｓｔの各桁が０～９以外の数を含んでいた場合、つまりＡＤＤＤＸ，ＳＵＢＤＸのオペランドがＢＣＤでなかった場合には、ＥＩＴとはならないが、ｄｅｓｔやフラッグに設定される結果は保証できない（インプリメント依存になる）ものとする。不正オペランド例外（ＩＯＥ）としないのは、インプリメントの負担が大きく、実行速度に影響が出るためである。

【０１８２】〔プログラム例外〕
・予約命令例外
・RR='11' のとき
・MM='11' のとき
・EaR が@-SPのとき
・EaM が#imm__data,@SP+,@-SPのとき
・〈〈Ｌ１〉〉機能例外
・ＡＤＤＤＸの正しいビットパターンがデコードされたとき
〔ニモニック〕
ＳＵＢＤＸ　ｓｒｃ，ｄｅｓｔ（本発明装置ではサポートしない）
〔命令の機能〕
dest - src - X__flag ==> dest BCD
１０進ＢＣＤの減算
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１７５に示す。
〔フラッグ変化〕図１７６に示す。
〔解説〕パックされたＢＣＤの減算を行なう。８ビット（２桁）、１６ビット（４桁）、３２ビット（８桁）、

147

６４ビット（１６桁）のＢＣＤデータを扱うことができる。ただし、６４ビットは〈〈ＬＸ〉〉仕様である。ソースオペランドのサイズがデスティネーションオペランドのサイズよりも小さい時は、ソースがゼロ拡張された上で加算される。ＢＣＤの数は符号拡張が無意味なので、基本的には符号なしの数と考え、ＳＵＢＤＸのフラッグ変化はＳＵＢＵに準じるものとする。結果が負になった時にＶ＿ｆｌａｇがセットされること、ｄ＜ｓの時はｄｅｓｔのサイズからの桁下げがＸ＿ｆｌａｇにセットされること、などもＳＵＢＵと同様である。ただし、ＳＵＢＵとは異なり、Ｚ＿ｆｌａｇはＡＤＤＸ，ＳＵＢＸのように累積で変化する。ＳＵＢＤＸで結果が負になった場合には、ｄｅｓｔは絶対値表現ではなく補数表現（１０の補数）となる。したがって、ｄｅｓｔは上位桁からの繰り下がりがあった場合と同じ値になる。

例：　16ビットでSUBDX を実行する場合

dest　src

0123 - 0456 = (-0333)

destは(-333) = 9667 となる。

ｓｒｃ，ｄｅｓｔの各桁が０～９以外の数を含んでいた場合、つまりＡＤＤＤＸ，ＳＵＢＤＸのオペランドがＢＣＤでなかった場合には、ＥＩＴとはならないが、ｄｅｓｔやフラッグに設定される結果は保証できない（インプリメント依存になる）ものとする。不正オペランド例外（ＩＯＥ）としないのは、インプリメントの負担が大＊

148

＊きく、実行速度に影響が出るためである。

【０１８３】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・MM='11' のとき

・EaR が@-SPのとき

・EaM が#imm__data.@SP+.@-SPのとき

・〈〈Ｌ１〉〉機能例外

・ＳＵＢＤＸの正しいビットパターンがデコードされたとき

〔ニモニック〕

ＰＡＣＫｓｓ　ｓｒｃ，ｄｅｓｔ（本発明装置ではサポートしない）

〔命令の機能〕

ｐａｃｋ　ｓｔｒｉｎｇ　ｉｎｔｏ　ＢＣＤ

ＢＣＤへのパック

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１７７に示す。

〔フラッグ変化〕図１７８に示す。

〔解説〕ｓｒｃをＢＣＤ（Ｂｉｎａｒｙ　Ｃｏｄｅｄ　Ｄｅｃｉｍａｌ）にパックしてｄｅｓｔに転送する。実際には、ｓｓにＢ，Ｈ，Ｗ，Ｌのいずれかの文字が入り、次のようなニモニックとオペレーションになる。

```
PACKHB  src[.H],dest[.B]
        RR=01,WW=00     src[04:07]==> dest[00:03],src[12:15]
                                  ==> dest[04:07]
PACKWH  src[.W],dest[.H]    <<L2>>

        RR=10,WW=01     src[04:07]==> dest[00:03],src[12:15]
                                  ==> dest[04:07]src[20:23]
                                  ==> dest[08:11],src[28:31]
                                  ==> dest[12:15]
PACKWB  src[.W],dest[.B]
        RR=10,WW=00     src[12:15]==> dest[00:03],src[28:31]
                                  ==> dest[04:07]
PACKLW  src[.L],dest[.W]  <<LX>>
PACKLH  src[.L],dest[.H]  <<LX>>
```

なお、ＰＡＣＫｓｓ，ＵＮＰＫｓｓにおいて、サイズの違いによってニモニックまで変えているのは、サイズの違いによって命令自体の意味もかなり変わると考えられるためである。つまり、一般の命令では、サイズの違いによってゼロ拡張や符号拡張を行なうだけであったが、ＰＡＣＫｓｓ，ＵＮＰＫｓｓでは命令のオペレーション自体がかなり異なっている。ＰＡＣＫ，ＵＮＰＫで、上記の説明に含まれない不合理なサイズの組み合わせを指定した場合には、動作を保証しない（インプリメントに依存した値がｄｅｓｔに設定される）ものとする。アーキテクチャ上は予約命令例外（ＲＩＥ）とするのが望ましいが、２つのオペランドのサイズの組み合わせによって予約命令例外（ＲＩＥ）を検出するのはインプリメントの負担が大きいため、予約命令例外（ＲＩＥ）とはし

149

ない。これは、異種サイズ間の論理演算の場合も同様である。ｓｒｃのうち、ｄｅｓｔに影響を与えないフィールド（ＰＡＣＫＨＢの２＾７～２＾４のビットなど）については、０かどうかのチェックは行なわず、０でなくても無視する。文字コードをそのままパックするケースを考えると、０でないことの方が多い。

【０１８４】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・W='11'のとき

・EaR が@-SPのとき

・EaW が#imm_data.@SP+ のとき

・〈〈Ｌ１〉〉機能例外

・ＰＡＣＫｓｓの正しいビットパターンがデコードされたとき

〔ニモニック〕

ＵＮＰＫｓｓ　ｓｒｃ，ｄｅｓｔ，ａｄｊ（本発明装置ではサポートしない）

〔命令の機能〕

ｕｎｐａｃｋ　ＢＣＤ

ＢＣＤからのアンパック

〔命令オプション〕なし

図１７９に示す。

〔フラッグ変化〕図１８０に示す。

〔解説〕ＢＣＤ（Ｂｉｎａｒｙ　Ｃｏｄｅｄ　Ｄｅｃｉｍａｌ）のｓｒｃのアンパックを行ない、アンパックした値に補正値ａｄｊを加えてｄｅｓｔに転送する。補正値ａｄｊを加えるのは、ＵＮＰＫ命令によって直接文字コードまで生成するためである。ａｄｊの加算は、ＢＣＤではなくバイナリの加算である。ａｄｊのサイズはｄｅｓｔのサイズと共通にWWフィールドによって指定される。実際には、ｓｓにＢ，Ｈ，Ｗ，Ｌのいずれかの文字が入り、次のようなニモニックとオペレーションになる。図１８１に示す。ＰＡＣＫ，ＵＮＰＫで、上記の説明に含まれない不合理なサイズの組み合わせを指定した場合には、動作を保証しない（インプリメントに依存した値がｄｅｓｔに設定される）ものとする。アーキテクチャ上は予約命令例外（ＲＩＥ）とするのが望ましいが、２つのオペランドのサイズの組み合わせによって予約命令例外（ＲＩＥ）を検出するのはインプリメントの負担が大きいため、予約命令例外（ＲＩＥ）とはしない。ａｄｊの加算によるオーバーフローは無視する。

【０１８５】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・W='11'のとき

・EaR が@-SPのとき

・EaW が#imm_data.@SP+ のとき

150

・〈〈Ｌ１〉〉機能例外

・ＵＮＰＫｓｓの正しいビットパターンがデコードされたとき

【０１８６】１２－１０．ストリング命令

「ストリング」とは、８ビット、１６ビット、３２ビット、または６４ビットのデータを任意の長さだけ連続して並べたデータタイプである。（ＳＳＣＨ命令に限り、連続ではなく、一定間隔で飛び飛びのアドレスにあるデータの集合もサポートしている。）

個々のデータの意味は特に決まっておらず、実際の文字コードになる場合、整数になる場合、浮動小数になる場合などがある。これは、ユーザ側で解釈する。ストリングの範囲を示す方法には、

・ストリングの長さ（データ数）を指定する方法

・ストリング終了を示す文字（ターミネータ）を指定する方法

の２通りがあり、使用目的や言語によって適した方を選択する必要がある。本発明装置のストリング命令ではストリングのデータ数がパラメータとなっているが、さらにオプションの終了条件という形でターミネータを与えることができ、両方の指定方法をサポートしている。本発明装置のストリング命令の特徴の一つとして、ポインタの増減量を自由に設定できるということがあげられる。そのため、ストリングサーチ命令（ＳＳＣＨ命令）を使ってテーブル検索や多次元配列のスキャンなども行なうことができる。また、本発明装置ではストリング命令ＳＭＯＶ，ＳＣＭＰ，ＳＳＣＨの終了条件として大小比較や二値比較を含む豊富な条件が指定でき、これも大きな特徴となっている。ストリング命令のうち、ストリングサーチ用のＳＳＣＨ命令は、検索条件が終了条件として指定されるため、終了条件にのみ意味がある命令となっている。本発明装置のストリング命令で指定できる条件（ｅｅｅｅ）については、付録を参照のこと。本発明装置では〈〈Ｌ２〉〉仕様となっている、終了条件ＯＵＴＵ－ＺＥはサポートしない。ストリング命令の用途としては、文字どおり８／１６ビットの文字列を処理するもののほか、特定のビットパターンのサーチ、メモリのブロック転送、構造体の代入、メモリ領域のクリアなどの応用がある。ストリング命令は任意長ビットフィールド命令と同じく不定長のデータを扱うため、実行中の割り込み受付、実行再開の機能が不可欠である。一方、ストリング命令自体がコンパイラの生成するコードとなることはほとんどなく、アセンブラで書かれたサブルーチンとして提供されることが多い。そのため、対称性やアドレッシングモードについての制限はあまり問題にならない。したがって、本発明装置のストリング命令では、オペランドや実行途中の状態保持のために固定番号のレジスタ（Ｒ０－Ｒ４）を使うようになっている。主なレジスタの使い方は次のようになる。

151　　152

R0:　ソース側ストリングの先頭アドレス
R1:　デスティネーション側ストリングの先頭アドレス
R2:　ストリングの長さ、データ数
R3:　終了条件の比較値(1)
R4:　終了条件の比較値(2)

このうち、ストリングの長さを表わすＲ２はエレメント数であって、バイト数ではない。Ｒ２は符号なしの数として扱われ、Ｒ２＝０の場合はエレメント数による命令終了は行なわないという意味に解釈される。つまり、エレメント数による終了を避けたい場合には、Ｒ２＝０として命令を実行すれば良いことになる。インプリメント上は、ストリング命令の実行パターンは次のようになるのが一般的である。

```
do {
        ...
        R2 - 1 ==> R2:
        check_interrupt;
} while (R2 !=0);
```

ただし、Ｒ２＝０とした場合に、エレメント数をH'１００００００００と考えるか、無限大（エレメント数の＊チェックを行なわない）と考えるかはインプリメントに依存する。つまり、H'１０００００００回のエレメント操作を行なっても命令を終了しなかった場合に、その後の動作はインプリメント依存になるものとする。ただし、エレメント数以外の要因により命令を終了した場合（Ｒ２＝０の場合は普通こうなる）には、命令終了後のＲ２の値（付録１１参照）が正しくセットされていなければならない。実際には、ＳＳＣＨ／ＲでＲ５＝０を指定したような特殊な場合を除いて、H'１０００００００回のエレメント操作を行なう間にアドレス変換例外（ＡＴＲＥ）やバスアクセス例外（ＢＡＥ）を起こし、命令中断となるのが普通である。ストリング命令はいろいろな要因で終了するため、それらを区別するためにフラッグを用いる。それぞれのフラッグの意味は、次のようになっている。

V__flag　エレメント数（ストリング長）による終了
F__flag　終了条件(eeee)による終了
この時、複数の終了条件を区別するために M__flagを使用する。
M__flagの変化については付録参照。

これ以外に終了要因のないＳＣＭＰ，ＳＳＣＨでは、Ｖ__ｆｌａｇとＦ__ｆｌａｇの変化が相補的となる。ＳＣＭＰの場合には、これ以外に比較データの不一致によって命令を終了する場合がある。

〔ニモニック〕　※

/F　アドレス増加の方向に処理する
/B　アドレス減少の方向に処理する

/ 終了条件各種(eeee)　（本発明装置では <<L2>> となっているOUTU-ZE は サポートしない。）

※ＳＭＯＶ

〔命令の機能〕
ｃｏｐｙ　ｓｔｒｉｎｇ
ストリングのコピー
〔命令オプション〕

〔命令ビットパターンとアセンブラ表記〕図１８２に示す。
〔フラッグ変化〕図１８３に示す。
〔解説〕ストリングの転送を行なう。減少方向の操作をするストリング命令ＳＭＯＶ／Ｂでは、最初にＲ０，Ｒ１で指定するアドレスが、操作対象となるストリングの占めるアドレスの最大アドレス＋１を指し、Ｒ０，Ｒ１をプリデクリメントしながら操作を進める。ＳＭＯＶでｓｒｃとｄｅｓｔがオーバーラップしていた時に、／Ｆ，／Ｂのうち正しくない方のオプションを指定した場合の動作は、ＢＶＣＰＹ，ＢＶＭＡＰと同じように保証しないものとする。つまり、インプリメントや命令実行中断の有無によって異なる場合がある。これは、高機能命令の特徴を生かしてパイプライン的なメモリアクセスをした場合に、メモリアクセスの順番が変わることがあり、必ずしも前エレメントの書き込みが終わってから次エレメントの読み出しを行なうとは限らないからである。なお、逆方向の処理オプション／Ｂは、この命令ＳＭＯＶ／Ｂに限り〈〈Ｌ２〉〉ではなく、〈〈Ｌ１〉〉となっている。任意長ビットフィールド命令やストリン

153

グ命令などの高機能命令の詳細仕様や、命令終了後のレジスタ値については、付録１１を参照のこと。

・予約命令例外
- SS='11' のとき
- P='1' のとき
- Q='1' のとき
- eeee='0111' ～'1111'のとき

〔ニモニック〕　※ｃｏｍｐａｒｅ　ｓｔｒｉｎｇ

ＳＣＭＰ　10　ストリングの比較

〔命令の機能〕　※　〔命令オプション〕

/F　アドレス増加の方向に処理する

/B　アドレス減少の方向に処理する

<<L2>>（本発明装置ではサポートする）

/ 終了条件各種(eeee)　（本発明装置では <<L2>> となっているOUTU-ZB はサポートしない。）

〔命令ビットパターンとアセンブラ表記〕図１８４に示す。

〔フラッグ変化〕図１８５に示す。

〔解説〕ストリングｓｒｃ１，ｓｒｃ２，の比較を行なう。２つのストリングが一致している間は比較を続け、一致しない文字が見付かれば比較を終了する。ＳＣＭＰ命令では、ＣＭＰ命令と同様に、ｓｒｃ２－ｓｒｃ１の結果をもとにしてフラッグの設定を行なう。例えばＬ＿ｆｌａｇは、ｓｒｃ１に対してｓｒｃ２の方がより小さ★

154

＊【０１８７】〔プログラム例外〕　＊

★いということを示す。ｓｒｃ１－ｓｒｃ２の結果をもとにしてフラッグの設定を行なうのではない。終了条件を持つＳＣＭＰ命令の応用としては、テキストを一行単位で比較する（Ｒ３に改行の文字コードをセットしてＳＣＭＰ／ＥＱを実行）、数字が続く間だけ比較する、全角文字が続く間だけ比較する、といったものがある。ＳＣＭＰでは、命令を終了させる要因が次の３つ存在し、フラッグ変化によってそれらを見分けることができる。

1. エレメント（データ）数(R2)による終了

V__flag = 1

2. 終了条件による終了

F__flag = 1, 終了要因によって M__flagが変化

3. 比較中のデータの不一致による終了

Z__flag = 0, 比較結果によって L__flag, X__flagが変化

L__flagは、最後のデータを符号付きと見て比較した時の比較結果

X__flagは、最後のデータを符号なしと見て比較した時の比較結果

１～３の要因のうち、２と３のチェックを同時に行なうことは可能であるが、１の要因のチェックは、２，３とは別のフェーズで行なわれる。したがって、２と３が同時に成立することはあるが、１と２や１と３が同時に成立することはない。１，２，３の少なくても一つの終了要因が成立した場合にＳＣＭＰ命令が終了する。比較するデータが一致している間は、その値（ｓｒｃ１＝ｓｒｃ２）が終了条件のテスト対象となるが、データに不一致があった場合には、Ｒ０により示されるｓｒｃ１の方を終了条件のテスト対象とする。ただし、不一致によりＳＣＭＰ命令が終了した場合には、終了条件が成立したかどうかという情報は必要ないことが多いため、これは単なる約束に過ぎない。また、終了条件が満たされないと意味を持たないＭ＿ｆｌａｇについては、別の終了要因により終了した場合に、意味が不明確となる。そのような場合のフラッグ変化は、０になるものと決めておく。Ｚ＿ｆｌａｇ，Ｌ＿ｆｌａｇ、Ｘ＿ｆｌａｇについては、一致、不一致にかかわらず必ず最後のデータの比較結果を反映する。したがって、３．以外の要因で終了した場合（データが一致している場合）には、自動的にＺ＿ｆｌａｇ＝１，Ｌ＿ｆｌａｇ＝０，Ｘ＿ｆｌａｇ＝０となる。なお、ＳＣＭＰでは符号なしデータ、符号付きデータの両方を扱うため、Ｌ＿ｆｌａｇにエレメント

155

を符号付きデータと考えた時の比較結果が入り、X__ｆｌａｇにエレメントを符号なしデータと考えた時の比較結果が入るようになっている。ＢＴＲＯＮの文字コードは符号なしで扱う必要があるし、一般の整数を扱うので＊

156

＊あれば、符号付きのデータを扱う必要もある。ＳＣＭＰのフラッグ変化をまとめると図１８６のようになる。○は終了要因が満たされたことを示し、×は終了要因が満たされないことを示す。

+ --- フラッグ本来の意味ではなく、約束としてこうなっていること
を示すものである

*A --- src1=src2 が終了条件のどれに該当するか、による

*B --- src1が終了条件のどれに該当するか、による

*C --- src1< src2 かsrc2< src1 か、による

／Ｂオプションは〈〈Ｌ２〉〉であるが本発明装置ではサポートする。

【０１８８】〔プログラム例外〕

・予約命令例外

・SS='11' のとき

・P='1' のとき

・Q='1' のとき

・eeee='0111' ～'1111'のとき

※〔命令の機能〕

ｆｉｎｄ　ａ　ｃｈａｒａｃｔｅｒ　ｉｎ　ａ　ｓｔｒｉｎｇ

ストリングのサーチ

〔命令オプション〕

/F　アドレス増加の方向に処理する

（ポインタ値をエレメントサイズずつ増加させる）

/R　ポインタの増加値はR5で指定される

〔ニモニック〕

ＳＳＣＨ

20

※

/ 終了条件各種(eeee)　（本発明装置では<<L2>>となっているOUTU-ZE はサポートしない。）

〔命令ビットパターンとアセンブラ表記〕図１８７に示す。

〔フラッグ変化〕図１８８に示す。

〔解説〕ストリングをサーチし、条件に合うエレメントを見つけ出す。'／Ｒ'オプションの場合には、Ｒ５の正負にかかわらず、常にエレメントの比較後にＲ０が更新（ポストインクリメントまたはポストデクリメント）★

30

★される。ＳＳＣＨ／ＲのＲ５のサイズはＲ０のポインタサイズと等しくなる。つまり、本発明装置３２では３２ビット固定であり、本発明装置６４ではＰビットまたはモードによって指定される。ＳＳ（Ｒ３，Ｒ４，エレメントサイズ）とは独立である。

【０１８９】〔プログラム例外〕

・予約命令例外

- SS='11' のとき

- P='1' のとき

- eeee='0111' ～'1111'のとき

〔ニモニック〕

ＳＳＴＲ

〔命令の機能〕

ｓｔｏｒｅ　ｃｈａｒａｃｔｅｒｓ

40

同一データを繰り返し書き込み（ストリングのフィル）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図１８９に示す。

〔フラッグ変化〕図１９０に示す。

〔解説〕先頭アドレス（Ｒ１）と長さ（Ｒ２）により指定されたメモリ領域に、Ｒ３の値を繰り返し書き込む。ＳＳＴＲ命令では終了条件が意味を持たないため、終了条件の指定は行なわない。なお、ストリング命令でＲ２＝０の場合はエレメント数による終了を行なわないが、

50

ＳＳＴＲ命令の場合はエレメント数による終了が唯一の終了要因であり、Ｒ２＝０を指定すると無限ループを形成することになる。これについては、ハードウエアでは特に対処せず、プログラム側で注意してもらうことにする。ただ、実行中の割り込み受付や再実行は可能なので、この命令によって間違って無限ループに入ったとしても、タスクやプロセスのスケジューリング等には影響しない。通常は複数の命令によって構成される無限ループが、たまたま一命令にまとめられただけと考える。Ｒ２＝０を不正オペランド例外ＩＯＥとしないのは、他のストリング命令との仕様の統一や、インプリメントの負担、高速化を考えたためである。このほか、ＳＳＣＨやＱＳＣＨ命令でも、パラメータや終了条件の指定によっては一つの命令で無限ループを形成する場合がある。

157

【０１９０】〔プログラム例外〕
・予約命令例外
・SS='11' のとき
・P='1' のとき

【０１９１】１２－１１．キュー命令
本発明装置ではキューを操作するための命令として、ＱＩＮＳ（エントリの挿入）、ＱＤＥＬ（エントリの削除）、ＱＳＣＨ（エントリのサーチ）の３つの命令が用意されている。本発明装置でサポートしているキューは、各キューエントリの先頭から１番目と２番目のデータが絶対アドレスのリンクポインタとなった、ダブルリンクのキューである。キューエントリの先頭にあるデータが次のキューエントリへのポインタとなり、キューエントリの２番目にあるデータが前のキューエントリへのポインタとなっている。キュー命令の仕様は、キューヘッダをキュー命令のオペランドとして直接指定できるように、以下のような方針で決められている。
１．ＱＤＥＬでは、指定したエントリではなく直後のエントリが削除される。ＱＵＥＵＥ　ＨＥＡＤをオペランドとして指定した場合には、先頭のエントリが削除されることになる。ＱＳＣＨ／Ｂで見付けたエントリを削除する場合や、キューの最後のエントリを削除する場合には間接参照が必要になるが、ＱＳＣＨ／Ｆで見付けたエントリを削除する場合やキューの先頭のエントリを削除する場合に比較すれば、頻度は少ないと考えられる。
２．ＱＩＮＳでは、指定したエントリの直前に新しいエントリが挿入される。ＱＵＥＵＥ　ＨＥＡＤをオペランドとして指定した場合には、キューの最後にエントリが挿入されることになる。これについては２つの考え方がある。ＱＤＥＬ命令との対称性を考えると、ＱＩＮＳでは指定したエントリ（あるいはＱＵＥＵＥ　ＨＥＡＤ）の直後にエントリを挿入する方が望ましい。これは、ＱＩＮＳで挿入したエントリをＱＤＥＬで削除するために、同じオペランドを指定できるからである。また、キューをスタック（ＬＩＦＯ）的な使い方にする場合にも、このような仕様の方が良い。一方、キューをＦＩＦＯで使う場合には、ＱＩＮＳでキューの最後にエントリを挿入し、ＱＤＥＬではキューの先頭のエントリを削除することが多い。こちらの方がキュー本来の使い方であるし、ＩＴＲＯＮでもそのような例がある。したがって、後者の仕様にする。
３．ＱＳＣＨでは、現在のエントリではなく直後のエントリからサーチを始める。ＱＵＥＵＥ　ＨＥＡＤをオペランドとして指定した場合には、キューの先頭からサーチを始めることになる。また、サーチが成功した場合に次のサーチを行なうには、そのままもう一度ＱＳＣＨを実行すればよい。この考え方は、他の高機能命令（ストリング、任意長ビットフィールド操作）とは異なったものである。すなわち、ストリング命令では現在ポインタ

158

の指しているデータ自体からサーチが始まり、連続サーチを行なう場合には別命令でポインタを更新しておく必要がある。これはキュー命令とは異なった動きである。しかし、キューの場合にはヘッダが別になっているという事情があり、扱いが異なっているため、別の仕様にしても構わないと判断した。
４．空のキューをフラッグで判定する。ＱＩＮＳで空のキューにデータを挿入した場合、ＱＤＥＬでエントリの削除の結果キューが空になった場合には、Ｚ＿ｆｌａｇをセットする。また、ＱＤＥＬで空のキューからエントリを削除しようとした場合は、エラーなのでポインタの付け変えは行なわないが、この時Ｖ＿ｆｌａｇをセットする。
〔ニモニック〕
ＱＩＮＳ　ｅｎｔｒｙ，ｑｕｅｕｅ
〔命令の機能〕
ｉｎｓｅｒｔ　ａ　ｎｅｗ　ｅｎｔｒｙ　ｉｎｔｏ　ａ　ｑｕｅｕｅ
ダブルリンクのキューへ挿入
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１９１に示す。
〔フラッグ変化〕図１９２に示す。
〔解説〕ｅｎｔｒｙで指定される新しいキューのエントリが、ｑｕｅｕｅで示されるキューエントリの直前に挿入される。ｑｕｅｕｅで指定されるキューエントリがキューヘッダであった場合には、この命令によって、キューの最後に新しいエントリが挿入されることになる。命令実行前にキューが空であったかどうかによって、Ｚ＿ｆｌａｇがセットされる。
［３２ビットで処理を行なう場合のＱＩＮＳ命令のオペレーション］図１９３に示す。
［実行前］図１９４に示す。
［実行後］図１９５に示す。ＥａＭｑＰ，ＥａＭａＰ２で指定されるアドレッシングモードでは、レジスタ直接Ｒｎ、@－ＳＰ，@ＳＰ＋，＃ｉｍｍ＿ｄａｔａのモードは使用できない。なお、ＱＩＮＳでは、命令の実行のために直接必要ではない部分のデータ構造のチェック（ｑｕｅｕｅの前後のキューエントリのリンク関係など）は特に行なわない。オペレーションに書かれている通りの動作を行なう。
【０１９２】〔プログラム例外〕
・予約命令例外
・+='0' のとき
・-='1' のとき
・EaMqP がRn, #imm_data, @SP+, @-SPのとき
・EaMqP2がRn, #imm_data, @SP+, @-SPのとき

〔ニモニック〕
ＱＤＥＬ　ｑｕｅｕｅ，ｄｅｓｔ

159

〔命令の機能〕
ｒｅｍｏｖｅ　ａｎ　ｅｎｔｒｙ　ｆｒｏｍ　ａ　ｑｕｅｕｅ
ダブルリンクのキューエントリを削除
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図１９６に示す。
〔フラッグ変化〕図１９７に示す。
〔解説〕ｑｕｅｕｅで指定されたキューエントリの次のエントリを削除し、削除されたエントリのアドレスをｄｅｓｔにセットする。削除されたエントリのアドレスをｄｅｓｔにセットするのは、それ以後削除したエントリを操作することが多いためである。ｑｕｅｕｅとしてキューヘッダを指定した場合には、キューの先頭のエントリが削除されることになる。ｑｕｅｕｅで指定されたキューが空のキューであった場合は、命令の実行ができない。この時、ＥＩＴは起動せず、Ｖ＿ｆｌａｇ，Ｚ＿ｆｌａｇのセットだけを行なって命令を終了する。ｄｅｓ＊

160

＊ｔは無変化となる。ｄｅｓｔ／ＥａＷ！Ｓでは、@－ＳＰのモードを禁止している。これは、キューが空でＶ＿ｆｌａｇがセットされ、ｄｅｓｔの転送ができない場合に、ｄｅｓｔに@－ＳＰが指定されていると命令動作がまぎらわしくなるためである。
［３２ビットで処理を行なう場合のＱＤＥＬ命令の　オペレーション］図１９８に示す。
［実行前］図１９９に示す。
［実行後］図２００に示す。ＥａＲｑＰで指定されるアドレッシングモードでは、レジスタ直接、@－ＳＰ，@ＳＰ＋，＃ｉｍｍ＿ｄａｔａのモードは使用できない。なお、ＱＤＥＬでは、空のキューの判定以外のチェック、命令の実行のために直接必要ではない部分のデータ構造のチェック（ｑｕｅｕｅの前後のキューエントリのリンク関係など）は特に行なわない。オペレーションに書かれている通りの動作を行なう。
【０１９３】〔プログラム例外〕

・予約命令例外
・t='0' のとき
・W='1' のとき
・EaRqP がRn, #imm_data,@SP+,@-SPのとき
・EaW!S が#imm_data,@SP+,@-SPのとき

〔ニモニック〕
ＱＳＣＨ
〔命令の機能〕
※ｓｅａｒｃｈ　ｑｕｅｕｅ　ｅｎｔｒｉｅｓ
キューのサーチ
※　〔命令オプション〕
/NM　R6 のマスクなし
/MR　R6 のマスクあり　<<L2>>（本発明装置ではサポートしない）
/F　キューの順方向にサーチ
/B　キューの逆方向にサーチ <<L2>> （本発明装置ではサポートする）

/ 終了条件各種(eeee)　（本発明装置では<<L2>>となっているOUTU-ZE はサポートしない）

〔命令ビットパターンとアセンブラ表記〕図２０１に示す。セットの必要なのは、Ｒ０，Ｒ２，Ｒ３（オプション），Ｒ４（オプション），Ｒ５，Ｒ６（オプション）であり、結果が入るのは、Ｒ０，Ｒ１である。続けて次のサーチを行なうことができる。
〔フラッグ変化〕図２０２に示す。
〔解説〕キューのエントリをサーチし、条件に合ったものを見付ける。逆方向のサーチ機能／Ｂ、およびマスクの機能／ＭＲは〈〈Ｌ２〉〉となっている。本発明装置では逆方向のサーチ機能／Ｂをサポートする。マスクの機能／ＭＲはサポートしない。この命令はキューの長さに依存した処理量となるので、ストリング命令と同じように実行中の中断に対する考慮が必要である。したがって、オペランドと途中の実行状態は固定番号のレジスタに置く。サーチ条件としては、マスク（特定ビットの抽出）と比較が用意されている。マスクはフラッグのサーチに用い、比較は優先度の処理などに用いる。比較条件の指定は、ストリング命令の終了条件の指定と同じである。キューの終りを判定するために、キューのエントリアドレスとキューの終了アドレスＲ２との比較を行ない、一致した場合には命令を終了する。Ｒ２との比較によって命令を終了した場合、すなわち、それまでにサーチ条件を満たすものがなく、サーチ失敗であった場合には、Ｖ＿ｆｌａｇをセットして命令を終了する。ＥＩＴは起動しない。ＱＳＣＨ命令の条件指定によっては、一つの命令の中で無限ループに入ることがある。これについては、ハードウエアでは特に対処せず、プログラム側で注意してもらうことにする。ただ、実行中の割り込み

161

受付や再実行は可能なので、ユーザプログラムの中で間違って無限ループに入ったとしても、タスクやプロセスのスケジューリング等には影響しない。通常は複数の命令によって構成される無限ループが、たまたま一命令にまとめられただけと考える。サーチが終了した時に、Ｒ０は指定した条件に合うエントリを、Ｒ１はその直前のエントリを指している。Ｒ１は、シングルリンクのキューの時にエントリを削除するために使用することができる。また、ＱＤＥＬでは指定したエントリの次のエントリが削除されるので、ＱＳＣＨ／Ｆで見付けたエントリ自体を削除する場合には、ＱＳＣＨ実行後、@Ｒ０ではなく@Ｒ１をパラメータとしてＱＤＥＬを実行すればよい。一般に、Ｒ０，Ｒ２にＱＵＥＵＥ　ＨＥＡＤのアドレスをセットしてＱＳＣＨ命令を実行することにより、キューが空の場合を含めてキュー全体のサーチを行なうことができる。ＱＳＣＨは、シングルリンクキューとダブルリンクキューで共用することを狙った命令である。

［ＱＳＣＨのオペレーション］図２０３に示す。このうち、ｃｈｅｃｋ＿ｉｎｔｅｒｒｕｐｔは、外部から割り込みがかかっているかどうかを調べ、割り込みがかかっていれば、ＱＳＣＨの実行を中断して割り込み処理を始めるというものである。割り込み処理終了後にＱＳＣＨ命令の残りの部分を実行する。

［実行前］図２０４に示す。

［実行後］図２０５に示す。

【０１９４】［プログラム例外］

・予約命令例外

・SS='11' のとき

・eeee='0111' ～'1111'のとき

・m='1' のとき

【０１９５】１２－１２．ジャンプ命令

〔ニモニック〕

ＢＲＡ　　ｎｅｗｐｃ

〔命令の機能〕

ｂｒａｎｃｈ　ａｌｗａｙｓ

ジャンプ（ＰＣ相対）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２０６に示す。

〔フラッグ変化〕図２０７に示す。

〔解説〕ＢＲＡ命令は、ＰＣ相対のみのアドレッシングをサポートするジャンプ命令である。ディスプレースメントのサイズとして、ＢＲＡ：Ｄでは８ビットが、ＢＲＡ：Ｇでは８ビット、１６ビット、３２ビット、６４ビットが利用できる。本発明装置の命令は必ず偶数アドレスから始まるので、短縮形のＢＲＡ：Ｄ命令では、＃ｄ８を２倍して使用する。すなわち、

PC + #d8 * 2 ==> PC

162

となる。ＢＲＡ：ＧでＳＳ＝００を指定した場合には、＃ｄＳを２倍せずにそのまま使用する。ＢＲＡ：Ｇでｎｅｗｐｃが１６ビットの場合、ＪＭＰ@（＃ｄＳ：１６，ＰＣ）と命令機能、コードサイズともに同じであるが、実行サイクル数を短くできる可能性があるため、別命令となっている。ＢＲＡ：Ｇで、ｎｅｗｐｃが奇数であった場合には、ジャンプ先が奇数アドレスになるため、奇数アドレスジャンプ例外（ＯＡＪＥ）となる。これは、Ｂｃｃ：Ｇ，ＢＳＲ：Ｇ，ＪＭＰ，ＪＳＲ命令も同様である。ＢＲＡ：Ｄ，Ｂｃｃ：Ｄ，ＢＳＲ：Ｄでは、オペランドを２倍して使用するため、ＯＡＪＥは発生しない。ＢＲＡ：Ｇ，Ｂｃｃ：Ｇ，ＢＳＲ：ＧでＳＳ＝００の場合、オペランドサイズは８ビットであるが、＃ｄＳフィールドは１６ビットとなる。この時、＃ｄＳフィールドは下位８ビットのみを使用し、上位８ビットには必ず０を入れておかなければならない。上位８ビットが０でない場合は、これによって表現されるデータがインプリメント依存の不定値になるものとする。つまり、ＢＲＡ：Ｇ命令の場合は、ジャンプ先が不定となる。ＥＩＴにはしない。本発明装置ではこの命令に対し、動的ブランチ予測処理をする。

【０１９６】［プログラム例外］

・予約命令例外

・SS='11' のとき

・P='1' のとき

・奇数アドレスジャンプ例外

・奇数アドレスにジャンプしたとき

〔ニモニック〕

Ｂｃｃ　　ｎｅｗｐｃ

〔命令の機能〕

ｂｒａｎｃｈ　ｃｏｎｄｉｔｉｏｎａｌｌｙ

条件ジャンプ（ＰＣ相対）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２０８に示す。

〔フラッグ変化〕図２０９に示す。

〔解説〕Ｂｃｃ命令は、ＰＣ相対のみのアドレッシングをサポートする条件ジャンプ命令である。ディスプレースメントのサイズとして、Ｂｃｃ：Ｄでは８ビットが、Ｂｃｃ：Ｇでは８ビット、１６ビット、３２ビット、６４ビットが利用できる。本発明装置の命令は必ず偶数アドレスから始まるので、Ｂｃｃ：Ｄ命令では、＃ｄ８を２倍して使用する。すなわち、

if(cccc)

PC + #d8 * 2 ==> PC

となる。Ｂｃｃ：ＧでＳＳ＝００を指定した場合には、＃ｄＳを２倍せずにそのまま使用する。Ｂｃｃの条件指

163

定部分（’ｃｃ’部分）の詳細とニモニック、ｃｃｃｃのビットパターンについては、付録を参照のこと。Ｂｃｃで未定義の条件を指定した場合には、予約命令例外（ＲＩＥ）となる。Ｂｃｃ：Ｇで条件不一致のためジャンプしなかった時は、本発明装置では奇数アドレスジャンプ例外（ＯＡＪＥ）を発生する場合と、発生しない場合がある。本発明装置ではこの命令に対し、動的ブランチ予測処理をする。

【０１９７】〔プログラム例外〕

・予約命令例外

・SS=’11’のとき

・P=’1’のとき

・cccc=’1110’～’1111’のとき

・奇数アドレスジャンプ例外

・奇数アドレスにジャンプしたとき

〔ニモニック〕

ＢＳＲ　ｎｅｗｐｃ

〔命令の機能〕

ｊｕｍｐ　ｔｏ　ｓｕｂｒｏｕｔｉｎｅ

サブルーチンジャンプ（ＰＣ相対）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２１０に示す。

〔フラッグ変化〕図２１１に示す。

〔解説〕ＢＳＲ命令は、ＰＣ相対のみのアドレッシングをサポートするサブルーチンジャンプ命令である。ＰＣの値がスタックに退避される。ディスプレースメントのサイズとして、ＢＳＲ：Ｄでは８ビットが、ＢＳＲ：Ｇでは８ビット、１６ビット、３２ビット、６４ビットが利用できる。本発明装置の命令は必ず偶数アドレスから始まるので、ＢＳＲ：Ｄ命令では、＃ｄ８を２倍して使用する。すなわち、

PC + #d8 * 2 ==> PC

となる。ＢＳＲ：ＧでＳＳ＝００を指定した場合には、＃ｄＳを２倍せずにそのまま使用する。ＢＳＲ，ＪＳＲ命令でスタックに退避されるＰＣ値としては、その次の命令の先頭アドレスを使用する。これに対して、実効アドレスの計算のためにＰＣを参照する場合（ＢＳＲなどで暗黙にＰＣを参照する場合を含む）には、その命令（次の命令ではない）の先頭アドレスをＰＣ値として使用するので、注意が必要である。ＢＳＲ，ＪＳＲでは旧のＰＣがスタックにセーブされるが、ＳＰのアラインメントに関しては特にチェックしない。ＳＰが４の倍数でない場合にも、そのまま実行される。本発明装置ではこの命令に対し、動的ブランチ予測処理をする。

【０１９８】〔プログラム例外〕

164

・予約命令例外

・SS=’11’のとき

・P=’1’のとき

・Q=’1’のとき

・奇数アドレスジャンプ例外

・奇数アドレスにジャンプしたとき

〔ニモニック〕

ＪＭＰ　ｎｅｗｐｃ

〔命令の機能〕

address of src ==> PC

ジャンプ

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２１２に示す。

〔フラッグ変化〕図２１３に示す。

〔解説〕ｎｅｗｐｃの実効アドレスにジャンプする。一般のアドレッシングモードが使用可能なジャンプ命令である。ｃａｓｅ文の実行などにおいては、ジャンプテーブルを参照してジャンプ先アドレスを決める場合がある。これはＪＭＰ命令と付加モードによるインデックスアドレッシングとを組み合せることにより実現する。

【０１９９】〔プログラム例外〕

・予約命令例外

・BaA がRn, #imm_data, @SP+, @-SPのとき

・奇数アドレスジャンプ例外

・奇数アドレスにジャンプしたとき

〔ニモニック〕

ＪＳＲ　ｎｅｗｐｃ

〔命令の機能〕

ｊｕｍｐ　ｔｏ　ｓｕｂｒｏｕｔｉｎｅ

サブルーチンジャンプ

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２１４に示す。

〔フラッグ変化〕図２１５に示す。

〔解説〕ｎｅｗｐｃの実効アドレスにサブルーチンジャンプする。ＰＣの値がスタックに退避される。ＢＳＲ，ＪＳＲ命令でスタックに退避されるＰＣ値としては、その次の命令の先頭アドレスを使用する。これに対して、実効アドレスの計算のためにＰＣを参照する場合（ＢＳＲなどで暗黙にＰＣを参照する場合を含む）には、その命令（次の命令ではない）の先頭アドレスをＰＣ値として使用するので、注意が必要である。

【０２００】〔プログラム例外〕

165

・予約命令例外
・P='1' のとき
・EaA がRn, #imm_data.@SP+.@-SPのとき

・奇数アドレスジャンプ例外
・奇数アドレスにジャンプしたとき
〔ニモニック〕
ＡＣＢ　ｓｔｅｐ，ｘｒｅｇ，ｌｉｍｉｔ，ｎｅｗｐｃ
〔命令の機能〕
ａｄｄ，ｃｏｍｐａｒｅ　ａｎｄ　ｂｒａｎｃｈ
インデクス値を増加するループ命令
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２１６に示す。
〔フラッグ変化〕図２１７に示す。
〔解説〕加算、比較、条件ジャンプを一命令にした複合命令であり、ループのプリミティブとして利用する。ｓｔｅｐ、ｘｒｅｇ、ｌｉｍｉｔは符号付き整数として演算、比較される。ｓｔｅｐは必ず正の値でないと条件ジャンプの意味がないが（ｘｒｅｇが終了値と反対の方向に変化することになる）、ｓｔｅｐの正負のチェックは＊

166

＊行なわず、オペレーションに書かれている通りの動作をそのまま行なう。ＡＣＢ命令では、ループ命令として高速実行ができるように、ｓｔｅｐ加算時のオーバーフローのチェックは行なわない。ｓｔｅｐを加算した結果オーバーフローが起こり、符号が反転した場合には、符号反転した正しくない値がそのままｌｉｍｉｔと比較される。ただし、比較のためのｌｉｍｉｔ－ｘｒｅｇの減算がオーバーフローしたとしても、ｘｒｅｇ＜ｌｉｍｉｔの比較は正確に行なわれる。ＡＣＢ，ＳＣＢではＰＣ相対でジャンプを行なう。ＳＳ＝００でディスプレースメントが８ビットになる場合も、ＳＳ≠００の場合と同様に、＃ｄＳ８は２倍せずにそのまま使用する。ＳＳ≠００の場合は、＃ｄＳ８のフィールドは使用せず（０にする）、ＳＳで指定されたサイズ（１６，３２，６４ビット）のデータが＃ｄＳ８の直後に続く。例えば、

```
ACB:Q   #1.R0. #4. label
```

で、ｌａｂｅｌとＡＣＢ：Ｑ命令のアドレスの差がH'１２３４であった場合は、図２１８に示すビットパターンになる。これは、固定長ビットフィールド命令の：Ｉフォーマットでも同じである。
［ＡＣＢのオペレーション］

```
Xreg + step ==> xreg
                    /* オーバーフローの場合は下位ビットのみ有効*/
if (xreg < limit) then PC + #dS8 ==>PC endif
```

ｎｅｗｐｃが奇数であった場合には、ジャンプ先が奇数アドレスになるため、奇数アドレスジャンプ例外（ＯＡＪＥ）となる。本発明装置では、終了条件満足のためジャンプしなかった時も、奇数アドレスジャンプ例外（ＯＡＪＥ）を発生する。
［詳細仕様調整中］ＡＣＢ，ＳＣＢ命令でＳＳ≠００の時には、＃ｄＳ８のフィールドは使用しない。この時、もし＃ｄＳ８のフィールドが０になっていなくても、単に無視される。ただし、マニュアル上は、＃ｄＳ８のフィールドには常に０を入れるようにしておく。本発明装置ではこの命令に対して、動的ブランチ予測処理をする。
【０２０１】〔プログラム例外〕
・予約命令例外
・RR='11' のとき
・XX='11' のとき
・SS='11' のとき
・P='1' のとき
・EaR が@-SPのとき
・EaRXが@-SPのとき

・奇数アドレスジャンプ例外
・奇数アドレスにジャンプしたとき
〔ニモニック〕
ＳＣＢ　ｓｔｅｐ，ｘｒｅｇ，ｌｉｍｉｔ，ｎｅｗｐｃ
〔命令の機能〕
ｓｕｂｔｒａｃｔ，ｃｏｍｐａｒｅ　ａｎｄ　ｂｒａｎｃｈ
インデクス値を減少するループ命令
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２１９に示す。
〔フラッグ変化〕図２２０に示す。
〔解説〕減算、比較、条件ジャンプを一命令にした複合命令であり、ループのプリミティブとして利用する。ｓｔｅｐ、ｘｒｅｇ、ｌｉｍｉｔなどは符号付き整数として比較される。ｓｔｅｐは必ず正の値でないと条件ジャンプの意味がないが、（ｘｒｅｇが終了値と反対の方向に変化することになる）。ｓｔｅｐの正負のチェックは行なわず、オペレーションに書かれている通りの動作をそのまま行なう。ＳＣＢ命令では、ループ命令として高速実行ができるように、ｓｔｅｐ減算時のオーバーフローのチェックは行なわない。ｓｔｅｐを減算した結果オーバーフローが起こり、符号が反転した場合には、符号反転した正しくない値がそのままｌｉｍｉｔと比較される。ただし、比較のためのｌｉｍｉｔ－ｘｒｅｇの減算

167

がオーバーフローしたとしても、ｘｒｅｇ＜ｌｉｍｉｔの比較は正確に行なわれる。ＡＣＢ，ＳＣＢではＰＣ相対でジャンプを行なう。ＳＳ＝００でディスプレースメントが８ビットになる場合も、ＳＳ≠００の場合と同様に、＃ｄＳ８は２倍せずにそのまま使用する。ＳＳ≠０＊

168

＊０の場合は、＃ｄＳ８のフィールドは使用せず（０にする）、ＳＳで指定されたサイズ（１６，３２，６４ビット）のデータが＃ｄＳ８の直後に続く。

［ＳＣＢのオペレーション］

```
xreg - step ==> xreg
/*オーバーフローの場合は下位ビットのみ有効*/
if (xreg ≧ limit) then PC + #dS8 ==>PC endif
```

ｎｅｗｐｃが奇数であった場合には、ジャンプ先が奇数アドレスになるため、奇数アドレスジャンプ例外（ＯＡＪＥ）となる。本発明装置では、終了条件満足のためジャンプしなかった時も、奇数アドレスジャンプ例外（ＯＡＪＥ）を発生する。

［詳細仕様調整中］ＡＣＢ，ＳＣＢ命令でＳＳ≠００の時には、＃ｄＳ８のフィールドは使用しない。この時、もし＃ｄＳ８のフィールドが０になっていなくても、単に無視される。ただし、マニュアル上は、＃ｄＳ８のフィールドには常に０を入れるようにしておく。本発明装置ではこの命令に対して、動的ブランチ予測処理をする。

【０２０２】〔プログラム例外〕

・予約命令例外

- RR='11' のとき
- XX='11' のとき
- SS='11' のとき
- P='1' のとき
- EaR が@-SPのとき
- EaRXが@-SPのとき

・奇数アドレスジャンプ例外

・奇数アドレスにジャンプしたとき

〔ニモニック〕

ＥＮＴＥＲ　ｌｏｃａｌ，ｒｅｇｌｉｓｔ

〔命令の機能〕

ｃｒｅａｔｅ　ｎｅｗ　ｓｔａｃｋ　ｆｒａｍｅ

スタックフレームの形成、高級言語用サブルーチンジャンプ

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２２１に示す。

〔フラッグ変化〕図２２２に示す。

〔解説〕高級言語用のスタックフレームを形成する。ＥＮＴＥＲのｌｏｃａｌは符号付きとして扱われ、ｌｏｃａｌのサイズが小さい場合には、ｌｏｃａｌの値が符号拡張される。ｌｏｃａｌが負の場合は意味がないスタックフレームが形成されるが、特にチェックは行なわず、オペレーションに書かれている通りの動作を行なう。この点は、ＡＣＢ，ＳＣＢのｓｔｅｐと同じである。

オペレーション：

```
FP -> ↓TOS
SP -> FP
SP - local -> SP
registers(mask) ->↓TOS
```

高級言語用のスタックフレームの詳細は、付録を参照のこと。退避するレジスタのビットマップ指定ＬｎＸＬは、図２２３のように行なう。ＬｎＸＬは、ＥａＲの拡張部よりも後に置かれる。ＥＮＴＥＲのｒｅｇｌｉｓｔでｂｉｔ０，ｂｉｔ１（ＳＰ，ＦＰ）を指定した場合には、単にその指定が無視されるものとする。ｂｉｔ０，ｂｉｔ１が”１”であっても、ＳＰ，ＦＰは転送されない。これを不正オペランド例外（ＩＯＥ）としないのは、インプリメントの負担が大きく、実効速度に影響が出るためである。ただし、チェックが行なわれていなくても、ＦＰ，ＳＰのビットには必ず’０’を入れてもらうように、マニュアル等で指導する必要がある。ＦＰ，ＳＰのアラインメントに関しては特にチェックしない。ＦＰ，ＳＰが４の倍数でない場合にも、オペレーションに書かれた通りの実行が行なわれる。ＥＮＴＥＲ：Ｇのｌｏｃａｌオペランドがメモリ上にあり、それがＥＮＴＥＲ命令の実行に伴って形成されるスタックフレーム領域と重なっていた場合には、命令再実行がきわめて難しくなる。そこで、ＥＮＴＥＲ：Ｇ，ＪＲＮＧ：Ｇ、および対称性からＥＸＩＴＤ：Ｇ命令では、メモリアクセスを伴うアドレッシングモード、つまりレジスタ直接Ｒｎとイミディエート以外のアドレッシングモードは、すべて禁止している。この命令のオペランドとして動的な値を設定したい場合には、テンポラリレジスタを一つ用意し、レジスタ直接Ｒｎのモードを利用するということになる。ｌｏｃａｌとしてＦＰ，ＳＰを指定した場合の動作は、インプリメント依存である。

【０２０３】〔プログラム例外〕

・予約命令例外
　・X='1' のとき
　・+='0' のとき
　・-='1' のとき
　・P='1' のとき
　SS='11' のとき
　・EaR!M が#imm__data,Rn 以外のとき

〔ニモニック〕
ＥＸＩＴＤ　ｒｅｇｌｉｓｔ，ａｄｊｓｐ
〔命令の機能〕
ｅｘｉｔ　ａｎｄ　ｄｅａｌｌｏｃａｔｅ　ｐａｒａｍｅｔｅｒｓ
高級言語用サブルーチンリターンとパラメータ解放
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２２４に示す。
〔フラッグ変化〕図２２５に示す。
〔解説〕高級言語用のスタックフレーム解放とレジスタの復帰を行ない、サブルーチンから戻る。その後ａｄｊｓｐをＳＰに加え、スタック上に残っていたサブルーチンのパラメータを捨てる。ＥＸＩＴＤのａｄｊｓｐは符号付きとして扱われ、ａｄｊｓｐのサイズが小さい場合には、ａｄｊｓｐの値が符号拡張される。ａｄｊｓｐが負の場合は意味のない動作をするが、特にチェックは行なわず、オペレーションに書かれている通りに実行を行なう。この点は、ＡＣＢ，ＳＣＢのｓｔｅｐと同じである。

オペレーション：

adjsp ==> tmp

↑TOS ==> registers(mask)
FP ==> SP
↑TOS ==> FP
↑TOS ==> PC
sp + adjsp ==> SP

高級言語用のスタックフレームの詳細は、付録を参照のこと。復帰するレジスタのビットマップ指定ＬｘＸＬは、図２２６のように行なう。ＬｘＸＬは、ＥａＲの拡張部よりも後に置かれる。ＥＸＩＴＤのｒｅｇｌｉｓｔでｂｉｔ１４，ｂｉｔ１５（ＳＰ，ＦＰ）を指定した場合には、単にその指定が無視されるものとする。ｂｉｔ１４，ｂｉｔ１５が”１”であっても、ＳＰ，ＦＰは転送されない。これを不正オペランド例外（ＩＯＥ）としないのは、インプリメントの負担が大きく、実効速度に影響が出るためである。ただし、チェックが行なわれていなくても、ＦＰ，ＳＰのビットには必ず’０’を入れてもらうように、マニュアル等で指導する必要がある。

ＦＰ，ＳＰのアラインメントに関しては特にチェックしない。ＦＰ，ＳＰが４の倍数でない場合にも、オペレーションに書かれた通りの実行が行なわれる。ＥＸＩＴＤで、スタックから復帰されたリターンアドレスが奇数であった場合には、ジャンプ先が奇数アドレスになるため、奇数アドレスジャンプ例外（ＯＡＪＥ）となる。ＥＸＩＴＤのオペランドａｄｊｓｐ／ＥａＲ！Ｍでは、メモリアクセスを伴うアドレッシングモード、つまりレジスタ直接Ｒｎとイミディエート以外のアドレッシングモードは、すべて禁止している。この命令のオペランドとして動的な値を設定したい場合には、テンポラリレジスタを一つ用意し、レジスタ直接Ｒｎのモードを利用するということになる。ａｄｊｓｐにレジスタ直接Ｒｎのモードを利用し、ｒｅｇｌｉｓｔに同じレジスタＲｎが含まれていた場合には、ａｄＪｓｐとして、レジスタ復帰前の値を使用する。つまり、スタック中に退避されていたＥＸＩＴＤ命令実行後のレジスタ値ではなく、ＥＸＩＴＤ命令実行前のレジスタ値がａｄＪｓｐとなる。ａｄｊｓｐとしてＦＰ，ＳＰを指定した場合の動作は、インプリメント依存である。

【０２０４】〔プログラム例外〕
・予約命令例外
　・X='1' のとき
　・+='0' のとき
　・-='1' のとき
　・P='1' のとき
　SS='11' のとき
　・EaR!M が#imm__data,Rn 以外のとき

〔ニモニック〕
ＲＴＳ
〔命令の機能〕
ｒｅｔｕｒｎ　ｆｒｏｍ　ｓｕｂｒｏｕｔｉｎｅ
サブルーチンからのリターン
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２２７に示す。
〔フラッグ変化〕図２２８に示す。
〔解説〕サブルーチンからのリターンを行なう。

171

オペレーション：

↑TOS -> PC

ＲＴＳで、スタックから復帰されたリターンアドレスが奇数であった場合には、ジャンプ先が奇数アドレスになるため、奇数アドレスジャンプ例外（ＯＡＪＥ）となる。

【０２０５】〔プログラム例外〕

・予約命令例外

・P='1' のとき

・奇数アドレスジャンプ例外

・リターンアドレスが奇数であったとき

〔ニモニック〕

ＮＯＰ

〔命令の機能〕

ｎｏ　ｏｐｅｒａｔｉｏｎ

ノーオペレーション

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２２９に示す。

〔フラッグ変化〕図２３０に示す。

〔解説〕何もしない。

【０２０６】〔プログラム例外〕

・予約命令例外

・-='1' のとき

〔ニモニック〕

ＰＩＢ

〔命令の機能〕

ｐｕｒｇｅ　ｉｎｓｔｒｕｃｔｉｏｎ　ｂｕｆｆｅｒ

命令キャッシュやパイプラインの整合性をとる

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２３１に示す。

〔フラッグ変化〕図２３２に示す。

〔解説〕命令パイプライン、命令キュー、命令キャッシュなど、命令実行の高速化のためのバッファ類をすべてパージし、メモリ上に置かれた命令列とプロセッサの内部状態との整合性を保証する。この命令は、これから実行すべき命令コードが、以前（リセット時あるいは前回のＰＩＢ命令実行時）から変更されている可能性があるということを、プロセッサに通知するために使用する。本発明装置では、パイプラインや命令キュー、命令キャッシュの制御を簡単化するため、プログラムにより命令コードを書き換えることは許されていない。つまり、自分自信で書き換えを行なった命令コードをそのまま実行しようとしても、動作が保証されない。ところが、ＯＳの行なう処理をマクロ的に見ると、プログラムをロードしてからそれを実行するという流れがある。つまり、広

172

い範囲で見るとＯＳのプログラムにより命令コードを書き換えていることになる。また、特殊な用途では、プログラムによって生成した命令列を実行することもある。この命令の目的は、そのような場合でも正しい命令の実行ができるようにすることである。すなわち、書き換えのあった命令コードに入る前にこの命令を実行しておけば、新しい命令コードが正しく実行されることが保証される。インプリメント上は、この命令によってパイプライン、命令キュー、命令キャッシュのパージを行なうことになる。ただし、パイプラインやキャッシュのメカニズムがメモリの書き換えに対するバスモニタリング機構を持っており、メモリとの整合性がハードウエアで常に保証されていれば、必ずしもＰＩＢ命令でパージを行なう必要はない。この場合、ＰＩＢ命令はＮＯＰ命令として実行される。いずれにしても、この命令を実行した後に、パイプラインや命令キャッシュとメモリとの整合性が保証されていれば良いのである。ＭＭＵを用いて多重論理空間を実現している場合には、ＰＩＢ命令を実行した論理空間に対してのみ書き換えた命令コードの実行が保証される。例えば、

```
context__A の命令コードを書き換え
STCTX
LDCTX  context__B
context__B の命令コードを書き換え
PIB
```

といった命令列を実行した場合、ｃｏｎｔｅｘｔ＿Ｂでは変更された命令コードを実行しても動作が保証されるが、次にＬＤＣＴＸ　ｃｏｎｔｅｘｔ＿Ａを実行した後でも、ｃｏｎｔｅｘｔ＿Ａの変更された命令コードの実行に対しては動作が保証されない。ｃｏｎｔｅｘｔ＿Ａの命令実行を保証するためには、ｃｏｎｔｅｘｔ＿Ａのコンテキストにおいて、もう一度ＰＩＢ命令を実行する必要がある。これは、命令キャッシュにＬＳＩＤが導入された場合に、ＰＩＢ命令では、ＬＳＩＤの一致する命令キャッシュエントリをパージするだけで済ませたいからである。ＰＩＢ命令以外の命令では、いかなるジャンプ命令やＯＳ関連命令（ＬＤＣＴＸ，ＲＥＩＴ，ＲＲＮＧ，ＴＲＡＰ，ＥＩＴ起動など）を実行した後でも、命令コード書き換え部分のプログラムの動作は保証されない。これは、命令キャッシュのパージをできるだけ減らすためである。したがって、ＯＳがロードしたプログラムを最初に実行する時には、新しいコンテキストに入ってから（例えばＬＤＣＴＸ～ＲＥＩＴの間で）、必ずＰＩＢを実行する必要がある。この命令のニモニックＰＩＢ（Ｐｕｒｇｅ　Ｉｎｓｔｒｕｃｔｉｏｎ　Ｂｕｆｆｅｒ）の’Ｂｕｆｆｅｒ’は、キャッシュやパイプラインなどを総括的に含めた意味で用いることばであり、ＰＴＬＢの’Ｂ’のＢｕｆｆｅｒに同じ用例がある。ＰＩＢというニモニックも、ＰＴＬＢとの連想から作られたも

173

のである。この命令は特権命令ではない。ユーザプログラムからも使用できる。

［命令コードの整合性について］ＰＩＢ命令の動作を正確に説明するため、「命令コードの整合性」という状態を以下のように定義する。「命令コードの整合性」とは、各論理空間の各論理アドレスについて、別々に定義される状態である。例えば、論理空間ＡではＨ’０００００００００～Ｈ’０００ｆｆｆｆｆについて「命令コードの整合性」が保証され、論理空間ＢではＨ’０００１０００００～Ｈ’０００３ｆｆｆｆについて「命令コードの整合性」が保証されている、といった使い方をする。「命令コードの整合性」が保証されている領域の命令を実行した場合にのみ、正しい命令動作をする（ｅｘｅｃｕｔｅのアクセス権チェックを含む）ことが保証される。一般には、「命令コードの整合性」の保証されている領域が命令コード領域であり、データ領域では「命令コードの整合性」が保証されていない。

・「命令コードの整合性」が保証されるようになるのは、次の場合である。

－リセット時

物理空間（＝論理空間）の全領域で「命令コードの整合性」が得られる。

－ＰＩＢ命令実行時

ＰＩＢ命令を実行した論理空間の全領域で「命令コードの整合性」が得られる。ＡＴ＝００の場合は、リセット時と同様、物理空間（＝論理空間）の全領域で「命令コードの整合性」が得られる。

・「命令コードの整合性」が失われるようになるのは、次の場合である。

－メモリ書き換え時

メモリ内容を書き換えた場合、書き換えた領域の「命令コードの整合性」は失われる。これは、論理アドレスによるメモリアクセスの場合も、物理アドレスによるメモリアクセスの場合（ＡＴ＝００やＬＤＰ命令など）も同様である。

－ＡＴＥ更新時

ＡＴＥを更新した場合、そのＡＴＥによりアドレス変換される領域の「命令コードの整合性」は失われる。したがって、例えば、ＬＤＡＴＥでＡＴＥ中の保護ビットを変更した場合にも、その後ＰＩＢ命令を実行しなければ保護情報のチェックが正しく行なわれないことになる。（これは、命令キャッシュと保護情報のチェックに関するインプリメントを軽くするために有効であろう）

以上の点に該当しない一般の命令実行ではＢＲＡ，ＪＭ

174

Ｐ，ＪＲＮＧ，ＲＲＮＧ，ＴＲＡＰ，ＲＥＩＴ，ＬＤＣＴＸ，ＥＩＴ起動などを含めて、「命令コード整合性の状態」は変化しない。

【０２０７】１２－１３．マルチプロセッサ用の命令

〔ニモニック〕

ＢＳＥＴＩ　ｏｆｆｓｅｔ，ｂａｓｅ

〔命令の機能〕

【０２０８】

【数１３】

```
~bit -> Z__flag, 1 -> bit (interlocked)
```

【０２０９】ビットのセット（バスをロック）

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２３３に示す。

〔フラッグ変化〕図２３４に示す。

〔解説〕指定されたビットの値を反転したものをＺ＿ｆｌａｇにコピーし、その後そのビットを１にセットする。この２つの操作はバスをロックして行なわれ、不可分の操作になる。したがって、マルチプロセッサ間の同期をとるためにこの命令が使用できる。ＳｈＭｆｑｉ，ＥａＭｆｉで指定されるアドレッシングモードでは、レジスタ直接モードＲｎ、@－ＳＰ，@ＳＰ＋，＃ｉｍｍ＿ｄａｔａのモードは使用できない。アセンブラ表記では、メモリアクセスのサイズをｂａｓｅのサイズとして指定する。ＢＳＥＴＩ：Ｑでは、メモリアクセスのサイズは８ビットに固定されており、サイズは’Ｂ’のみを書くことができる。また、ＢＳＥＴＩ：Ｇ，ＢＳＥＴＩ：Ｅでのアクセスサイズ（ｂａｓｅのサイズ）．Ｈ，．Ｗの指定は、ＢＳＥＴ，ＢＣＬＲと同じく〈〈Ｌ２〉〉とする。〈〈Ｌ２〉〉仕様でアクセスサイズ．Ｈ，．Ｗを指定したのに、ｂａｓｅがアラインメントのとれていないアドレスであった場合には、メモリアクセスの範囲がインプリメント依存となる。これは、ビット操作命令と同様である。この時、インプリメントによって、アラインメントのとれていないワードやハーフワードのアクセスが行なわれる場合には、バスをロックしたまま複数のバスサイクルを実行する。これはＣＳＩ命令と同様である。本発明装置では〈〈Ｌ２〉〉となっているハーフワード、ワード単位のアクセスのインプリメントを行なう。またｂａｓｅとしてアラインメントのとれていないアドレスを指定した場合にも、アラインメントのとれたハーフワード、ワード単位でアクセスを行なう。

【０２１０】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・BB='11' のとき

・EaR が@-SPのとき

・EaMfi,ShMfqiがRn, #imm__data,@SP+,@- SPのとき

175

〔ニモニック〕
ＢＣＬＲＩ　ｏｆｆｓｅｔ，ｂａｓｅ
〔命令の機能〕
【０２１１】
【数１４】
```
~bit -> Z__flag, 0 -> bit (interlocked)
```
【０２１２】ビットのクリア（バスをロック）
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２３５に示す。
〔フラッグ変化〕図２３６に示す。
〔解説〕指定されたビットの値を反転したものをＺ＿ｆｌａｇにコピーし、その後そのビットを０にセットする。この２つの操作はバスをロックして行なわれ、不可分の操作になる。したがって、マルチプロセッサ間の同期をとるためにこの命令が使用できる。ＥａＭｆｉで指定されるアドレッシングモードでは、レジスタ直接モードＲｎ、@－ＳＰ，@ＳＰ＋，＃ｉｍｍ＿ｄａｔａのモードは使用できない。アセンブラ表記では、メモリアクセスのサイズをｂａｓｅのサイズとして指定する。ＢＣＬＲＩ：Ｇ，ＢＣＬＲＩ：Ｅでのアクセスサイズ（ｂａｓｅのサイズ）．Ｈ，．Ｗの指定は、ＢＳＥＴ，ＢＣＬＲと同じく〈〈Ｌ２〉〉とする。〈〈Ｌ２〉〉仕様でアクセスサイズ．Ｈ，．Ｗを指定したのに、ｂａｓｅがアラインメントのとれていないアドレスであった場合には、メモリアクセスの範囲がインプリメント依存となる。これは、ビット操作命令と同様である。この時、インプリメントによって、アラインメントのとれていないワードやハーフワードのアクセスが行なわれる場合には、バスをロックしたまま複数のバスサイクルを実行する。これはＣＳＩ命令と同様である。本発明装置では〈〈Ｌ２〉〉となっているハーフワード、ワード単位のアクセスのインプリメントを行なう。またｂａｓｅとし＊

176

＊てアラインメントのとれていないアドレスを指定した場合にも、アラインメントのとれたハーフワード、ワード単位でアクセスを行なう。
【０２１３】〔プログラム例外〕
・予約命令例外
- ・RR='11' のとき
- ・BB='11' のとき
- ・EaR が@-SPのとき
- ・EaMfi がRn，＃imm__data,@SP+,@-SPのとき

〔ニモニック〕
ＣＳＩ　ｃｏｍｐ，ｕｐｄａｔｅ，ｄｅｓｔ
〔命令の機能〕
ｃｏｍｐａｒｅ　ａｎｄ　ｓｔｏｒｅ（ｉｎｔｅｒｌｏｃｋｅｄ）
比較とストア（バスをロック）
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２３７に示す。
〔フラッグ変化〕図２３８に示す。
〔解説〕ｄｅｓｔの値が以前の値（ｃｏｍｐにより指定）と同じであれば、内容を更新する命令である。この命令は、簡単な構造のデータをマルチプロセッサから更新していく場合に利用できる。ＣＳＩ命令を実行した結果、ｄｅｓｔの値が以前の値と異なっていることがわかった場合、それは他のプロセッサがデータ内容を書き換えたことを意味している。したがって、ＣＳＩ命令によってｄｅｓｔの値の食い違いを発見したプロセッサは、新しいｄｅｓｔの値をもとにして、そのデータ内容の更新をやり直さなければならない。このような方法をとることにより、マルチプロセッサの下でデータの一貫性を保つことができる。［ＣＳＩのオペレーション］

```
update ==> tmp
/*以下の動作はバスをロックして行なわれる*/
if (dest = comp)
then
        tmp ==> dest
        1 ==>  Z__flag
else
        dest ==> comp
        0 ==>  Z__flag
```

ビットパターン上の制約から、ＣＳＩでは、比較が成功しなくてもｕｐｄａｔｅの読みだしが行なわれる。また、ＣＳＩ命令でのｄｅｓｔのアクセス権は、常にｒｅａｄ，ｗｒｉｔｅとも必要であるものとする。すなわちＣＳＩ命令で比較が失敗し、ｄｅｓｔに対して書き込みが起らない場合でも、ｄｅｓｔに対してｗｒｉｔｅアクセス権がないとアドレス変換例外（ＡＴＲＥ）になる。ＲＭＣ，ＥａＭｉＲのサイズはＲＲで指定される。ＥａＭｉＲで指定されるアドレッシングモードでは、@－ＳＰ，@ＳＰ＋，Ｒｎ，＃ｉｍｍ＿ｄａｔａのモードは使用できない。ＣＳＩ命令で、サイズ．Ｈ，．Ｗを指定し、アラインメントの取れていないアドレスをオペラン

177

ドとした場合には、バスをロックしたまま複数のバスサイクルを実行する。この場合、ｒｅａｄ，ｗｒｉｔｅのそれぞれが２回ずつのメモリアクセスに分かれるので、命令全体では、バスをロックしながらｒｅａｄ，ｒｅａｄ，ｗｒｉｔｅ，ｗｒｉｔｅの４回のメモリアクセスを＊

```
var1  EQU  H'00000006  ：アラインメントの取れていないアドレス
```

とした場合に、プロセッサＡから

```
MOV.W  #H'12345678.@var1
```

を実行し、プロセッサＢから

```
MOV.W  #H'87654321.@var1
```

を実行すると、メモリ書き込みのタイミングによっては、

H'00000006～7　=H'8765
H'00000008～9　=H'5678

となって、プロセッサＡのＭＯＶ命令が先に実行された場合ともプロセッサＢのＭＯＶ命令が先に実行された場※

178

＊行なうことになる。なお、ＣＳＩ以外の一般命令で、アラインメントの取れていないアドレスに対してメモリアクセスを行なった場合には、バスはロックされない。したがって、例えば、

※合とも異なる結果になる可能性がある。マルチプロセッサ間の共有変数に対しては、通常データの書き込みだけではなくデータ更新（ｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅ）を行なうのが普通なので、必然的にＣＳＩ命令を使うことになり、以上のような問題は発生しない。しかし、マルチプロセッサからＣＳＩ以外の命令でアラインメントのとれていない変数をアクセスする場合には、以上のような問題が生じることがあるので、注意しておく必要がある。

【０２１４】〔プログラム例外〕

・予約命令例外
・RR='11' のとき
・EaR が@-SPのとき
・EaMiR がRn. #imm__data.@SP+,@-SPのとき

【０２１５】１２－１４．制御空間、物理空間操作命令

本発明装置では、メインプロセッサの制御レジスタ群が、コプロセッサの制御レジスタ群やチップバス上の高速メモリなどとともに一つのアドレス空間を作ることができるようになっており、これを制御空間と呼ぶ。制御空間の考え方は、現在別チップとなっているコプロセッサやコンテキスト退避用の高速メモリが、将来メインプロセッサに内蔵された場合に、特に有効になる考え方である。制御レジスタ操作命令は、制御空間に対してアクセスを行なうための命令である。なお、ＬＤＣ，ＳＴＣなどの汎用的な制御空間操作命令は特権命令となっているため、ユーザが制御空間の一部であるＰＳＢ，ＰＳＭを操作するためには、ＬＤＰＳＢ，ＳＴＰＳＢ，ＬＤＰＳＭ，ＳＴＰＳＭ命令を使用する。本発明装置はアドレス変換機構を持たない。よって、論理空間アドレスと物理空間アドレスがつねに等しいため物理空間操作命令の機能は論理空間を操作する他の命令に吸収されてしまう。しかし、アドレス変換機構を持ち、論理空間と物理空間を区別して扱う本発明装置とのソフトウエア互換性を重視し、本発明装置では物理空間操作命令をサポートする。

〔ニモニック〕

ＬＤＣ　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｌｏａｄ　ｃｏｎｔｒｏｌ　ｓｐａｃｅ　ｏｒ　ｒｅｇｉｓｔｅｒ

制御空間へのロード

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２３９に示す。

〔フラッグ変化〕図２４０に示す。

〔解説〕ｓｒｃの値を制御空間のｄｅｓｔに転送する。ｓｒｃのサイズがｄｅｓｔより小さいときは、符号拡張される。ｄｅｓｔ／ＥａＷ%では、レジスタ直接モードＲｎの指定、@－ＳＰの指定はできない。この命令は特権命令である。ｒｉｎｇ０以外から実行された場合には、特権命令違反例外（ＲＩＶＥ）となる。本発明装置では制御空間に対する．Ｂ，．Ｈのアクセス機能はサポートしない。制御空間としてはＣＰＵ内の制御レジスタのみをインプリメントする。また、ＵＡＴＢ，ＳＡＴＢを実装していないためＬＤＣによりＵＡＴＢ，ＳＡＴＢを変更することはできない。ＬＤＡＴＥ，ＳＴＡＴＥ，ＬＤＰ，ＳＴＰ，ＬＤＣ，ＳＴＣ，ＭＯＶＰＡ命令の中の特殊空間を参照するオペランドにおいて、付加モードによりメモリの間接参照が起こった場合には、特殊空間の方ではなく論理空間（ＬＳ）の方を参照する。また、スタックポインタＳＰの参照があった場合には、ＰＲＮＧではなく現在リングＲＮＧのスタックが参照される。特殊空間のアドレスという意味を持つのは、最終的に得られた実効アドレスのみである。制御空間に対する．Ｂ，．Ｈのアクセス機能が全くないプロセッサにおいて、制御空間のオペランドのサイズとして．Ｂ，．Ｈを指定した場合には、予約命令例外（ＲＩＥ）となる。未実装の制御レジスタ、または制御レジスタのないアドレスをＬＤＣで指定した場合には、予約機能例外（ＲＦＥ）となる。〈〈ＬＶ〉〉の領域についても同様であ

179

る。制御空間で利用できるアドレスに何らかの制限のあるプロセッサの場合、それに違反した場合には予約機能例外（ＲＦＥ）とする。例えば、制御レジスタのアドレスを４の倍数に限るといった制限はこれに含まれる。コンテキスト退避用の高速メモリを内蔵したプロセッサであれば、制御レジスタ部分のアドレスのみが４の倍数に制限され、高速メモリ部分のアドレスは自由になるというケースが考えられるが、この場合にも、違反すると予約機能例外（ＲＦＥ）になる。また、一部のアドレスについてのみ．Ｂ，．Ｈの指定が可能なプロセッサにおいて、．Ｂ，．Ｈのアクセスができないアドレスを指定した場合にも、予約命令例外（ＲＩＥ）ではなく予約機能例外（ＲＦＥ）となる。これは、命令ビットパターン（サイズ指定を含む）のみでエラーと判定できるものを予約命令例外（ＲＩＥ）とし、アドレスやオペランド値によってエラーかどうかの状態が変化するものは予約機能例外（ＲＦＥ）とする、という考え方に基づいたものである。制御空間のアドレスがチップ外（コプロセッサのアドレスなど）になり、インプリメントの制約によってその領域がアクセスできなくなっている場合にも、予約機能例外（ＲＦＥ）が発生する。ＬＤＣ，ＳＴＣでは、制御空間のアドレスがコプロセッサのアドレスになった場合でも、コプロセッサ命令例外（ＣＩＥ）は発生しない。コプロセッサ命令例外（ＣＩＥ）が発生するのは、コプロセッサ用の命令を実行した場合に限られる。ＬＤＣで、制御レジスタの’－’，’＋’で表現されるｒｅｓｅｒｖｅｄのビットに異なる値を書き込もうとした場合や、あるフィールドに対してｒｅｓｅｒｖｅｄの値を書き込もうとした場合には、予約機能例外（ＲＦＥ）になる。ＰＳＷのＳＭＲＮＧのフィールドに’００１’などのｒｅｓｅｒｖｅｄの値を書き込んだ場合も、これに含まれる。一方、’＝’，’＃’で表現されているｒｅｓｅｒｖｅｄのビットに異なる値を書き込んだ場合には、単に無視される。ただし、ユーザ向けのマニュアルでは、’＝’に対して必ず’０’を書き込んでもらうように注意しなければならない。また、’＊’で表現されているビットには、何を書き込んでも単に無視される。このビットは、’＝’，’＃’とは異なり、今後仕様を拡張した場合でも、使用されないことが保証されたビットである。したがって、ＬＤＣを実行する前に、このビットを’０’にマスクしておく必要はない。ＬＤＣでＣＴＸＢＢを変更した場合には、メモリ上のＣＴＸＢＢの内容と実際のチップ内のコンテキストとの整合性がとれなくなるが、これはプログラマの責任で処理する。ハードウエア的には、単にＣＴＸＢＢの変更のみを行なう。ＣＴＸＢＢの変更とコンテキストのロードを両方行なう場合は、ＬＤＣＴＸを使用すればよい。ＬＤＣ命令によってＵＡＴＢ，ＳＡＴＢが変更される時は、それに伴ってＴＬＢや論理キャッシュのパージ（ＰＳＴＬＢ／ＡＴに相当する処理）が自動的に行なわれる。ＬＳＩＤ

180

を実装したプロセッサの場合は、ＬＳＩＤ制御レジスタにより指定される論理空間がパージの対象となる。この場合、ＬＤＣ命令には、ＰＳＴＬＢ命令のような／ＳＳ，／ＡＳのオプションは設けられていないが、これは次のような理由によっている。ＰＴＬＢ，ＰＳＴＬＢ命令によるＴＬＢのパージの場合は、ＬＤＣ　＊，ＵＡＴＢの場合とは異なり、他の論理空間のキャッシュやＴＬＢもパージできるように、ＬＳＩＤの機能に相当するパラメータを、別のレジスタ（Ｒ１）によって指定している。この場合、ＬＳＩＤの制御レジスタは使用しない。したがって、そのパラメータを使用するかどうかを区別するために、／ＳＳ，／ＡＳのオプションを切り換える必要がある。ところが、ＬＤＣ　＊，ＵＡＴＢの場合は、データの矛盾をなくするために、現在使用中の空間に対してキャッシュやＴＬＢのパージを行なうのであるから、ＬＳＩＤの制御レジスタは本来の意味で働く。つまり、一般のメモリアクセスと同様に、ＬＳＩＤ制御レジスタによって指定された論理空間がパージの対象となる。ＬＳＩＤ未実装のプロセッサでは、全論理空間（一つだけであるが）がパージの対象となる。

【０２１６】〔プログラム例外〕

・予約命令例外

- RR='11' のとき
- WW='10' 以外のとき
- EaR が@-SPのとき
- EaW%がRn, #imm_data,@SP+,@-SPのとき

・特権命令違反例外

・ｒｉｎｇ０以外から実行されたとき

・予約機能例外

・未実装の制御レジスタをアクセスしたとき

・制御レジスタの特定フィールドに対してｒｅｓｅｒｖｅｄの値を書き込もうとしたとき（＝，＃，＊は除く）

・ＥａＷ%のアドレスのワードアラインメントがとれていないとき

〔ニモニック〕

ＳＴＣ　　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｓｔｏｒｅ　ｃｏｎｔｒｏｌ　ｓｐａｃｅ　ｏｒ　ｒｅｇｉｓｔｅｒ

制御空間からのストア

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２４１に示す。

〔フラッグ変化〕図２４２に示す。

〔解説〕制御空間にあるｓｒｃの値をｄｅｓｔに転送する。ＳＴＣでは、ｓｒｃとｄｅｓｔのサイズが共通に指定されるため、異種サイズ間の転送は行なわれない。この命令は特権命令である。ｒｉｎｇ０以外から実行された場合には、特権命令違反例外（ＰＩＶＥ）となる。ｓ

181

ｒｃ／ＥａＲ%では、レジスタ直接モードＲｎの指定、イミディエート＃ｉｍｍ__ｄａｔａの指定、@ＳＰ＋の指定はできない。本発明装置では、制御空間に対する．Ｂ，．Ｈのアクセス機能はサポートしない。制御空間についてはＣＰＵ内の制御レジスタのみをインプリメントする。ＬＤＡＴＥ，ＳＴＡＴＥ，ＬＤＰ，ＳＴＰ，ＬＤＣ，ＳＴＣ，ＭＯＶＰＡ命令の中の特殊空間を参照するオペランドにおいて、付加モードによりメモリの間接参照が起こった場合には、特殊空間の方ではなく論理空間（ＬＳ）の方を参照する。また、スタックポインタＳＰの参照があった場合には、ＰＲＮＧではなく現在リングＲＮＧのスタックが参照される。特殊空間のアドレスという意味を持つのは、最終的に得られた実効アドレスのみである。制御空間に対する．Ｂ，．Ｈのアクセス機能が全くないプロセッサにおいて、制御空間のオペランドのサイズとして．Ｂ，．Ｈを指定した場合には、予約命令例外（ＲＩＥ）となる。未実装の制御レジスタ、または制御レジスタのないアドレスをＳＴＣで指定した場合には、予約機能例外（ＲＦＥ）となる。〈〈ＬＶ〉〉の領域についても同様である。制御空間で利用できるアドレスに何らかの制限のあるプロセッサの場合、それに違反した場合には予約機能例外（ＲＦＥ）とする。例えば、制御レジスタのアドレスを４の倍数に限るといった制限はこれに含まれる。コンテキスト退避用の高速メモリを内蔵したプロセッサであれば、制御レジスタ部分のアドレスのみが４の倍数に制限され、高速メモリ部分のアドレスは自由になるというケースが考えられるが、この場合にも、違反すると予約機能例外（ＲＦＥ）になる。また、一部のアドレスについてのみ．Ｂ，．Ｈの指定が可能なプロセッサにおいて、．Ｂ，．Ｈのアクセスができないアドレスを指定した場合にも、予約命令例外（ＲＩＥ）ではなく予約機能例外（ＲＦＥ）となる。これは、命令ビットパターン（サイズ指定を含む）のみでエラーと判定できるものを予約命令例外（ＲＩＥ）とし、アドレスやオペランド値によってエラーかどうかの状態が変化するものは予約機能例外（ＲＦＥ）とする、という考え方に基づいたものである。制御空間のアドレスがチップ外（コプロセッサのアドレスなど）になり、インプリメントの制約によってその領域がアクセスできなくなっている場合にも、予約機能例外（ＲＦＥ）が発生する。ＬＤＣ，ＳＴＣでは、制御空間のアドレスがコプロセッサのアドレスになった場合でも、コプロセッサ命令例外（ＣＩＥ）は発生しない。コプロセッサ命令例外（ＣＩＥ）が発生するのは、コプロセッサ用の命令を実行した場合に限られる。ＳＴＣで、制御レジスタの'－'で表現されているビットを読みだした場合には'０'が、'＋'のビットを読みだした場合には'１'が読み出される。また、'＝'，'＃'，'＊'のビットを読み出そうとした場合に得られる値は、不定である。インプリメントによって、'０'固定の場合、'１'固

182

定の場合、以前に書き込んだ値がそのまま読み出される場合がある。将来の拡張のため、'＝'，'＃'，'＊'のビットの値を利用したプログラミングは行なわないように、ユーザ向けのマニュアルに明記しなければならない。

【０２１７】〔プログラム例外〕

・予約命令例外

・WW='10' 以外のとき

・EaR%がRn, #imm__data,@SP+,@-SPのとき

・EaW が#imm__data,@SP+ のとき

・特権命令違反例外

・ｒｉｎｇ０以外から実行されたとき

・予約機能例外

・未実装の制御レジスタをアクセスしたとき

・ＥａＲ%のアドレスのワードアラインメントがとれていないとき

〔ニモニック〕

ＬＤＰＳＢ　　ｓｒｃ

〔命令の機能〕

ｌｏａｄ　ＰＳＢ

ＰＳＢへのロード

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２４３に示す。

〔フラッグ変化〕図２４４に示す。

〔解説〕ｓｒｃをＰＳＢに転送する。ユーザのコルーチンなどで、ＰＳＢ，ＰＳＭの個々のビットの意味とは関係なく退避や復帰を行なう場合を除けば、ＰＳＭ，ＰＳＢでは、一部のフィールドのみの書き換えをしたいということが多い。そのため、ＬＤＰＳＢ，ＬＤＰＳＭ命令のｓｒｃオペランドは１６ビット（ＥａＲｈ）となっており、上位バイトがマスク（変更されるビットを０とする）、下位バイトが変更データを表わすという仕様になっている。つまり、ｓｒｃを

src = [S0. S1...S7. S8. S9...S15]

とすると、

［ＬＤＰＳＢのオペレーション］

【０２１８】

【数１５】

([S0. S1...S7].and.PSB).or.(-[S0. S1...S7]

.and.[S8. S9...S15]) ==>PSB

ただし'－'はビット否定

【０２１９】となる。例えば、２＾４の位置にあるＸ__ｆｌａｇをセットする命令は、

LSPSB　#H'ef10

となる。上位バイトで、変更されるビットの方を０、変

183

更されない方を１としたのは、変更される方をデフォルトと考える方が自然だと判断したからである。８ビット全部を変更する場合には、上位バイトをすべて０にして単なるバイトデータを書けばよい。８ビット全部の変更は、最初に述べたように、ユーザ側でＰＳＢ，ＰＳＭの退避や復帰をする場合に必要である。ＬＤＰＳＢ，ＬＤＰＳＭで、ＰＳＢ，ＰＳＭの未使用フィールドの値を”１”にしようとした場合には、予約機能例外（ＲＦＥ）が発生する。

【０２２０】〔プログラム例外〕

・予約命令例外

・ＥａＲｈが@－ＳＰのとき

〔ニモニック〕

ＬＤＰＳＭ　　ｓｒＣ

〔命令の機能〕

ｌｏａｄ　ＰＳＭ

ＰＳＭへのロード

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２４５に示す。

〔フラッグ変化〕図２４６に示す。

〔解説〕ｓｒｃをＰＳＭに転送する。ユーザのコルーチンなどで、ＰＳＢ，ＰＳＭの個々のビットの意味とは関係なく退避や復帰を行なう場合を除けば、ＰＳＭ，ＰＳＢでは、一部のフィールドのみの書き換えをしたいということが多い。そのため、ＬＤＰＳＢ，ＬＤＰＳＭ命令のｓｒｃオペランドは１６ビット（ＥａＲｈ）となっており、上位バイトがマスク（変更されるビットを０とする）、下位バイトが変更データを表わすという仕様になっている。つまり、ｓｒｃを

src = [S0. S1... S7. S8. S9... S15]

とすると、

［ＬＤＰＳＭのオペレーション］

【０２２１】

【数１６】

([S0. S1... S7]. and. PSM). or. (~[S0. S1... S7]
. and. [S8. S9... S15])==>PSM

ただし ' ~ ' はビット否定

【０２２２】となる。ＬＤＰＳＢ，ＬＤＰＳＭで、ＰＳＢ，ＰＳＭの未使用フィールドの値を”１”にしようとした場合には、予約機能例外（ＲＦＥ）が発生する。

【０２２３】〔プログラム例外〕

・予約命令例外

・EaRhが@-SPのとき

〔ニモニック〕

ＳＴＰＳＢ　　ｄｅｓｔ

〔命令の機能〕

ｓｔｏｒｅ　ＰＳＢ

184

ＰＳＢからのストア

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２４７に示す。

〔フラッグ変化〕図２４８に示す。

〔解説〕ＰＳＢをｄｅｓｔに転送する。上位８ビットは必ず０となる。ｄｅｓｔが８ビットではなく１６ビットとなっており、上位８ビットが常に０を返すようになっているのは、ＬＤＰＳＭ，ＬＤＰＳＢでそのままＰＳＭ，ＰＳＢの復帰ができるように配慮したためである。

【０２２４】〔プログラム例外〕

・予約命令例外

・EaWhが#imm__data.@SP+ のとき

〔ニモニック〕

ＳＴＰＳＭ　　ｄｅｓｔ

〔命令の機能〕

ｓｔｏｒｅ　ＰＳＭ

ＰＳＭからのストア

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２４９に示す。

〔フラッグ変化〕図２５０に示す。

〔解説〕ＰＳＭをｄｅｓｔに転送する。上位８ビットは必ず０となる。ｄｅｓｔが８ビットではなく１６ビットとなっており、上位８ビットが常に０を返すようになっているのは、ＬＤＰＳＭ，ＬＤＰＳＢでそのままＰＳＭ，ＰＳＢの復帰ができるように配慮したためである。

【０２２５】〔プログラム例外〕

・予約命令例外

・EaWhが#imm__data,@SP+ のとき

〔ニモニック〕

ＬＤＰ　　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｌｏａｄ　ｐｈｙｓｉｃａｌ　ｓｐａｃｅ

物理空間へのロード

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２５１に示す。

〔フラッグ変化〕図２５２に示す。

〔解説〕ｓｒｃの値を物理空間のｄｅｓｔに転送する。ｓｒｃのサイズがｄｅｓｔより小さいときは、符号拡張される。本発明装置はアドレス変換機構を持たないので論理空間アドレスと物理空間アドレスがつねに等しく、この命令の機能はＭＯＶ命令に吸収されてしまう。しかし、アドレス変換機構をもち論理空間と物理空間を区別して扱う本発明装置とのソフトウエア互換性を取るためこの命令をサポートする。この命令は特権命令である。ｄｅｓｔ／ＥａＷ％では、レジスタ直接モードＲｎの指

185

定、@－ＳＰの指定はできない。ＬＤＡＴＥ，ＳＴＡＴＥ，ＬＤＰ，ＳＴＰ，ＬＤＣ，ＳＴＣ，ＭＯＶＰＡ命令の中の特殊空間を参照するオペランドにおいて、付加モードによりメモリの間接参照が起こった場合には、特殊空間の方ではなく論理空間（ＬＳ）の方を参照する。また、スタックポインタＳＰの参照があった場合には、ＰＲＮＧではなく現在リングＲＮＧのスタックが参照される。特殊空間のアドレスという意味を持つのは、最終的に得られた実効アドレスのみである。

【０２２６】〔プログラム例外〕

・予約命令例外

・RR='11' のとき

・WW='11' のとき

・EaR が@-SPのとき

・EaW%がRn, #imm__data.@SP+,@-SPのとき

・特権命令違反例外

・ｒｉｎｇ０以外から実行されたとき

〔ニモニック〕

ＳＴＰ　ｓｒｃ，ｄｅｓｔ

〔命令の機能〕

ｓｔｏｒｅ　ｐｈｙｓｉｃａｌ　ｓｐａｃｅ

物理空間からのストア

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２５３に示す。

〔フラッグ変化〕図２５４に示す。

〔解説〕物理空間にあるｓｒｃの値をｄｅｓｔに転送する。ＳＴＰでは、ｓｒｃとｄｅｓｔのサイズが共通に指定されるため、異種サイズ間の転送は行なわれない。本発明装置はアドレス変換機構を持たないので論理空間アドレスと物理空間アドレスがつねに等しく、この命令の機能はＭＯＶ命令に吸収されてしまう。しかし、アドレス変換機構を持ち論理空間と物理空間を区別して扱う本発明装置とのソフトウエア互換性を取るためこの命令をサポートする。この命令は特権命令である。ｓｒｃ／ＥａＲ%では、レジスタ直接モードＲｎの指定、イミディエート　＃ｉｍｍ__ｄａｔａの指定、@ＳＰ＋の指定はできない。ＬＤＡＴＥ，ＳＴＡＴＥ，ＬＤＰ，ＳＴＰ，ＬＤＣ，ＳＴＣ，ＭＯＶＰＡ命令の中の特殊空間を参照するオペランドにおいて、付加モードによりメモリの間接参照が起こった場合には、特殊空間の方ではなく論理空間（ＬＳ）の方を参照する。また、スタックポインタＳＰの参照があった場合には、ＰＲＮＧではなく現在リングＲＮＧのスタックが参照される。特殊空間のアドレスという意味を持つのは、最終的に得られた実効アドレスのみである。

【０２２７】〔プログラム例外〕

186

・予約命令例外

・WW='11' のとき

・EaR%がRn, #imm__data.@SP+,@-SPのとき

・EaW が#imm__data.@SP+ のとき

・特権命令違反例外

・ｒｉｎｇ０以外から実行されたとき

【０２２８】１２－１５．ＯＳ関連命令

〔ニモニック〕

ＪＲＮＧ　ｖｅｃｔｏｒ　（「本発明装置」ではサポートしない）

〔命令の機能〕

ｊｕｍｐ　ｔｏ　ｎｅｗ　ｒｉｎｇ

リング間コール

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２５５に示す。

〔フラッグ変化〕図２５６に示す。

〔解説〕リング間の遷移とジャンプ（リング間コール）を行なう。この命令は、現在のリングよりも内側のリングレベルにあるプログラムの呼び出し（システムコールの呼び出しを含む）を行なうために使用される。外側のリングから内側のリングを保護するため、ＪＲＮＧでのジャンプ先は特定のアドレスに制限されている。このアドレスを入れたテーブルをリング間遷移テーブルＪＲＮＧＶＴ（ＪＲＮＧ　Ｖｅｃｔｏｒ　ｔａｂｌｅ）と呼ぶ。ＪＲＮＧ命令では、ｖｅｃｔｏｒオペランドがＪＲＮＧＶＴに対するインデクスとなる。ＪＲＮＧＶＴの一つのエントリを、ＪＲＮＧＶＴＥと呼ぶ。ＪＲＮＧＶＴは、ｖｅｃｔｏｒに対するエントリを６５５３５個持つテーブルであり、そのベースの論理アドレスがＪＲＮＧＶＢによって示される。ｖｅｃｔｏｒのサイズは１６ビットとなっている。ＪＲＮＧＶＢは制御レジスタの一つであり、次のような構成になっている。

【０２２９】・ＪＲＮＧＶＢ

ＪＲＮＧＶＢは、図２５７に示すようにＪＲＮＧ命令のベクトルテーブルの先頭の論理アドレスを示す。テーブルのベースアドレスは、アラインメントのため、下位の３ビットが０に固定される。Ｅが０の時はＪＲＮＧは実行禁止であり、ＪＲＮＧを実行するとリング遷移違反例外（ＲＴＶＥ）となる。この時、ＪＲＮＧＶＢは意味を持たないので、ＯＳはこのフィールドを自由に使用して良い。’＝’のビットには’０’を入れておかなければならない。ただし、このビットが０でなくても単に無視される。ＪＲＮＧＶＴＥは８バイトであり、図２５８のような構成になっている。これは、内側のリングへ入るためのゲートという意味合いをもったものである。

・ＡＲの機能は、そのｖｅｃｔｏｒで示されるエントリのリング間コールが、最低でどのリングから発行可能で

187

あるかを示すものである。ＡＲで示されるリングよりも現在のリングの方が外側の場合には、リング間コール（システムコール）が許可されていないものと見なされ、リング遷移違反例外（ＲＴＶＥ）となる。ＡＲは、その意味から考えて、ＰＳＷのＰＲＮＧの位置に相当するフィールドを使う。これは、ＪＲＮＧＶＴ，ＥＩＴＶＴの各エントリが、基本的にはＰＳＷ＋ＰＣのサブセットの形になっているという考え方に基づいたものである。

・ＶＸの機能は、ＯＳとユーザプログラムとの間で３２／６４ビットのモードが異なっている場合に有効である。

・ＪＲＮＧＶＴＥの未使用フィールド（'＝'で示される）、および'ＶＸ'ビットには、'０'を入れておかなければならない。ただし、実際には、これらのビットが'１'であったとしても、単に無視されるだけである。これを予約機能例外（ＲＦＥ）としないのは、インプリメントの負担を考えたためである。

［詳細仕様調整中］

・また、ＪＲＮＧＶＴＥのＶＰＣのフィールドは偶数でなければならない。つまり、ＶＰＣのフィールドのＬＳＢは'０'でなければならない。違反した場合には、ＪＲＮＧ実行時に奇数アドレスジャンプ例外（ＯＡＪＥ）が発生する。［詳細仕様調整中］

【０２３０】ＪＲＮＧＶＢはＭＳＢ＝０の時ＵＡＴＢ、ＭＳＢ＝１の時ＳＡＴＢを使ってアドレス変換される。ＪＲＮＧＶＢのアドレスを論理アドレスとしたのは、次のような利点があるためである。

①テーブルをコンテキスト毎に持つことが可能

②テーブルの仮想化が可能である。すなわち、テーブル自体をページ不在にすることができる。

③ＥＩＴであるＴＲＡＰＡとの性格の違いが、よりはっきりする。

ＪＲＮＧＶＢを論理アドレスと考えることにより、テーブルの仮想化が可能となる。「本発明装置」では、ベクトルが１６ビット（６５５３６エントリ、５１２ＫＢテーブルサイズ）で非常に多くなっており、しかもベクトルの上限を指定するレジスタが設けられていない。しかし、ＪＲＮＧＶＢが論理アドレスとなっているため、ＭＭＵ機能との組合わせが可能であり、必ずしもテーブル分の物理メモリを負担する必要はなくなっている。ＪＲＮＧＶＴ部分のＳＴＥ，ＰＴＥを未使用領域にしておけば、１６ビット＝６５５３６エントリ分のテーブルをすべて物理メモリで用意する必要はない。ＪＲＮＧＶＴＥの読みだしは、論理アドレスを用いた一般のメモリアクセスと同様に行なわれる。したがって、ＪＲＮＧＶＴＥは、ＪＲＮＧを実行したプログラムのリングのアクセス権で読み出される。ＪＲＮＧを実行したリングから、指定したベクトルのＪＲＮＧＶＴＥをｒｅａｄする権利がない時は、アドレス変換例外（ＡＴＲＥ）のリング保護違反エラーになる。また、指定したベクトルのＪＲＮＧＶＴＥが未使用領域となっていた場合には、アドレス変換例外（ＡＴＲＥ）の未使用領域参照エラーとなる。使用する側から見ると、これらのエラーはリング遷移違反例外（ＲＴＶＥ）と同じように扱われる方が望ましいのであるが、主としてインプリメントの都合により、上記のような仕様になっている。ＪＲＮＧＶＴＥの読み出しの際には、このほかページ不在例外（ＰＯＥ）、バスアクセス例外（ＢＡＥ）などの発生する可能性がある。ＪＲＮＧ機能の導入により、ＪＲＮＧＶＴのための論理空間を必ず５１２ＫＢ消費することになる。また、不当なリング間コールを防ぐためには、ユーザプログラムを実行する前に、ＯＳがＪＲＮＧＶＴの領域のＳＴＥ，ＰＴＥのセットを行なっておく必要がある。そこで、リング間コール機能を使用しない場合に、こういった処理が全く不用になるように、リング間コール機能全体をディスエーブルにすることができるようになっている。この指定には、ＪＲＮＧＶＢのＬＳＢのＥビットを使う。ＪＲＮＧＶＢのＥビットが０の場合は、リング間コール機能は使用できなくなり、ＪＲＮＧを実行した場合には無条件にリング遷移違反例外（ＲＴＶＥ）となる。

188

【０２３１】結局、ＪＲＮＧが成功するには次の条件を満たさなければならない。

－－－ＪＲＮＧＶＢのＥ＝１。

Ｅ＝０であれば、ＪＲＮＧＶＴが用意されていないことを意味するのでリング遷移違反例外（ＲＴＶＥ）となる。

－－－指定したベクトルに対するＪＲＮＧＶＴＥが、ＪＲＮＧ実行前のリングからｒｅａｄ可能。

ページ不在例外が発生した場合は、ページインの後再実行することになる。アドレス変換例外（ＡＴＲＥ）の未使用領域参照エラーが発生した場合は、そこまでテーブルが用意されていないことを意味するので、そのユーザプログラムにエラーを返すのが普通である。ｒｅａｄのアクセス権がない場合は、保護の観点からＪＲＮＧの実行を禁止していることを意味するので、やはりそのユーザプログラムにエラーを返すのが普通である。これはＶＡフィールドと同等の意味になるが、５１２ベクトル毎の指定となる。

－－－現在リング≧ＶＲが成立。

外側のリングには行けない。違反した場合はリング遷移違反例外（ＲＴＶＥ）となる。

－－－現在リング≦ＡＲが成立。

受け付け可能リングのチェック。違反した場合はリング遷移違反例外（ＲＴＶＥ）となる。なお、ＡＲはＪＲＮＧＶＴＥのＡＲフィールドである。

【０２３２】［ＪＲＮＧのオペレーション］

189　　190

if JRNGVB のＥビット=0 then リング遷移違反例外(RTVE)

論理アドレスmem[vector×8+JRNGVB] よりVR,AR,VPC をフェッチ

if旧RNG > AR .or.旧RNG < VR thenリング遷移違反例外(RTVE)

旧SP ==> TOS↓（VRで示される新しいスタックを使用）

旧PC ==> TOS↓

　　旧PCとしては、JRNG命令の次の命令の先頭アドレスがスタックにプッシュされる。

　　これは、JSR 命令と同じである。

旧PSW .and. B'01110000__00000000__11111111__11111111 ==> TOS↓

　　旧PSW は、RRNGで意味を持つフィールド、つまり、RNG,XA,PSHのフィールドのみがそのままスタックにプッシュされ、それ以外のSM,AT,IMASK のフィールドは０にマスクされてからスタックにプッシュされる。これは、外側のリングのプログラムが、OSのみが知っているべき情報(IMASKなど）を読み出せないようにするためである。

旧RNG ==> 新PRNG

VR ==>新RNG

VPC ==> 新PC

ＪＲＮＧ命令により形成されるスタックフレームは、図２５９のようになる。

【０２３３】ここで、旧リングのＳＰを新リングのスタックに積んだのは、新リングから旧リングのスタックポインタＳＰやスタックをアクセスするためである。リング毎のスタックは制御レジスタとしてアクセス可能であるが、アクセスのためには特権命令（ＳＴＣ）を利用しなければならない。したがって、例えばｒｉｎｇ１からｒｉｎｇ３のスタックに積まれたパラメータを見たいという場合には、この機能が必要である。ＪＲＮＧでは、ＰＳＳの一部とＰＳＭのＰＲＮＧのみが更新され、ＰＳＢは更新されない。また、リング間コールの機能では、ＥＩＴとは異なり、スタックフォーマットは一種類しかないので、ＦＯＲＭＡＴ（ＥＩＴＩＮＦ）はスタックに積まれない。ＪＲＮＧ：Ｅの場合は、ｖｅｃｔｏｒがゼロ拡張される。ＡＴ＝００　（アドレス変換なし）の場合は、ＪＲＮＧＶＢは物理アドレスを表わす。ＪＲＮＧ実行時に、リング遷移違反例外（ＲＴＶＥ）などの命令再実行型のＥＩＴが発生した場合、ＪＲＮＧ固有の機能であるリング間コール用のスタックフレームの形成は行なわれず、ＥＩＴ処理用のスタックフレームのみが形成される。例えば、ＳＭＲＮＧ＝０００の状態でＪＲＮＧを実行し、ＲＮＧ＝００にジャンプしようとしたが、何らかのエラーによりＥＩＴが発生した場合は、図２６０のようなスタックフレームになる。図２６１ではない。（Ａ）のような仕様になっているのは、ＥＩＴ発生後の命令再実行のことを考えているからである。つまり、ＥＩＴ処理ハンドラに入る前に、プロセッサの状態をできるだけ命令実行前の状態に戻そうとしているからである。ＥＩＴで使用するスタックとＪＲＮＧで使用するスタックが異なっていた場合は、ＥＩＴ発生時にＥＩＴで使用するスタックのみが変化し、ＪＲＮＧで使用するスタックのＳＰは変化しない。

【０２３４】ＪＲＮＧでは、現在リングと同じリングにジャンプすることも可能である。この場合、ＪＲＮＧでスタックの切り換えは起こらない。ＳＰとしてスタックにプッシュされる値は、命令実行前のＳＰの値になる。これは、ＪＲＮＧ命令の最初でＰＵＳＨＳＰを実行したのと同じ動作である。この様子を図２６２に示す。ＪＲＮＧで現在リングと同じリングにジャンプする場合に、ＪＲＮＧ：Ｇのｖｅｃｔｏｒオペランドがメモリ上にあり、それがＪＲＮＧ命令の実行に伴って形成されるスタックフレーム領域と重なっていると、命令再実行がきわめて難しくなる。そこで、ＪＲＮＧ：Ｇ命令では、メモリアクセスを伴うアドレッシングモード、つまりレジスタ直接Ｒｎとイミディエート以外のアドレッシングモードは、すべて禁止している。この命令のオペランドとして動的な値を設定したい場合には、テンポラリレジスタを一つ用意し、レジスタ直接Ｒｎのモードを利用する必要がある。リング間コールの機能はＥＩＴには含めない。ＴＲＡＰＡとＪＲＮＧは、どちらもＯＳのシステムコールの呼出しを行なうことを目的にした命令である。一般には、ＢＴＲＯＮのようにシステムコールの数が多く、複数のリングを利用するＯＳではＪＲＮＧを利用し、ＩＴＲＯＮのようにシステムコールの数があまり多くなく、２リングで十分なＯＳではＴＲＡＰＡを利用す

191

ることになる。ＴＲＡＰＡでは、ｒｉｎｇ１，ｒｉｎｇ２に行くことはできないので、ＢＴＲＯＮで外核をｒｉｎｇ１に置く場合には、当然ＪＲＮＧを使う必要がある。

【０２３５】なお、ＯＳをユーザが拡張するような場合には、外向きのリング間コールが必要になることがあり、ＢＴＲＯＮでもそういった例がある。しかし、頻度＊

192

＊としては多くないこと、リング保護の趣旨とは矛盾していること、完全な保護を行なうためにはリターンスタックを別に設ける必要があり、インプリメントの負担が大きくなること、などの理由により、命令セットのレベルでは外向きのリング間コールをサポートしない。

【０２３６】〔プログラム例外〕

・予約命令例外
・P='1' のとき
・EaRh!MがRn, #imm__data以外のとき

・〈〈Ｌ１〉〉機能例外
・ＪＲＮＧの正しいビットパターンがデコードされたとき

〔ニモニック〕
ＲＲＮＧ　（「本発明装置」ではサポートしない）
〔命令の機能〕
ｒｅｔｕｒｎ　ｆｒｏｍ　ｐｒｅｖｉｏｕｓ　ｒｉｎｇ※

↑TOS ==> temp1
↑TOS ==> temp2
↑TOS ==> SP of temp1<RNG>
if RNG> temp1<RNG> thenリング遷移違反例外(RTVE)起動
・temp1<RNG>は、temp1 を PSWと見た時にRNG フィールドに相当する部分を表わす。
・このチェックがないと、RRNG命令で不正に内側のリングに入ることが可能になる。
temp1<PSH> ==> PSH (PRNGを含む)
temp1<RNG> ==> RNG
temp1<XA> ==> XA
temp2 ==> PC

※リング間リターン
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２６３に示す
〔フラッグ変化〕図２６４に示す
〔解説〕リング間コールに対するリターンを行なう。

【０２３７】［ＲＲＮＧのオペレーション］

このほか、ＲＲＮＧ命令実行時にはＤＣＥのＥＩＴが起動される場合があり、そのチェックを行なう必要がある。詳しくは付録９を参照。ここで、ＰＲＮＧの旧のスタックポインタをＲＮＧのスタックからポップして、再びＰＲＮＧのスタックポインタとして設定しているのは、ＰＲＮＧのスタックに積まれたシステムコールのパラメータのポップなどにより、ＯＳ側からユーザのスタックポインタを更新することを想定したためである。ＲＲＭＧで内側のリングに入ろうとした場合には、リング遷移違反例外（ＲＴＶＥ）となる。また、スタックからポップされたＰＣが奇数であった場合には、奇数アドレスジャンプ例外（ＯＡＪＥ）となる。現在のＰＳＷのＳＭが０で、ＲＲＮＧ命令によりポップされるスタック中のＲＮＧ（上記オペレーション中のｔｅｍｐ１＜ＲＮＧ＞）が０でない場合は、ＲＲＮＧ命令の実行によって、ＰＳＷ中のＳＭとＲＮＧの組み合せがｒｅｓｅｒｖｅｄのパターンとなる。この場合には、予約機能例外（ＲＦＥ）となる。

【０２３８】ＲＲＮＧ命令で、命令再実行型の例外であるＲＴＶＥ，ＲＦＥが発生した場合には、ＲＲＮＧ命令実行後に無くなる予定であったリング間コールのスタックフレームが、そのまま残る。したがって、ＥＩＴとリング間コールで同じスタックを使っていた場合には、そのスタックフレームに追加される形でＥＩＴのスタックフレームが形成される。また、ＥＩＴとリング間コールで異なるスタックを使っていた場合には、リング間コールで使用していたスタックの内容やスタックポインタは変化しない。この点は、ＲＲＮＧでＤＣＥを起動する場合とは異なっている。ＤＣＥの場合には、前のリング間コールのスタックフレームをクリアしてから、ＤＣＥの新しいスタックフレームを構成する。

〈〈ＲＦＥ発生時のスタックの例－ＥＩＴで同じスタックを利用する場合〉〉図２６５に示すこれに対して、ＯＡＪＥは命令完了型のＥＩＴとなる予定である。［詳細

193

仕様調整中]
その場合は、ＤＣＥと同じように、リング間コールのスタックフレームをクリアしてからＥＩＴのスタックフレームを生成するということになる。ＲＲＮＧ命令でＯＡＪＥが発生する場合のスタックの動きは、次のようになる。
〈〈ＯＡＪＥ発生時のスタックの例－ＥＩＴで同じスタックを利用する場合〉〉
（ＲＲＮＧ実行前）図２６６に示す
（ＲＲＮＧ実行後、ＯＡＪＥ起動後）図２６７に示す
ＲＲＮＧ命令でスタック中からポップされたＰＳＷ（上記のｔｅｍｐ１）のうち、ＰＳＨ，ＲＮＧ，ＸＡ以外のフィールドは、無視される。ただし、プログラミング上は、ＪＲＮＧ命令からＲＲＮＧ命令までの間で、スタック中に退避されたＰＳＷのＰＳＨ，ＲＮＧ，ＸＡ以外のフィールドを書き換えてはいけない。ＲＲＮＧ命令（３２ビット）で同じリングに戻る場合、ＳＰの最終値は

mem[initSP + 8] ==> SP

（ただし、+8はPC.PSWの分）

となる。これは、ＰＣ，ＰＳＷの処理の後ＰＯＰ　ＳＰを実行したのと同じ動作である。ＪＲＮＧＶＢのＥビットは、ＲＲＮＧ命令の動作には関係しない。Ｅビットが０の場合にも、ＲＲＮＧ命令の実行は行なわれる。
【０２３９】〔プログラム例外〕
・予約命令例外
・Ｐ＝’１’のとき
・〈〈Ｌ１〉〉機能例外
・ＲＲＮＧの正しいビットパターンがデコードされたとき
〔ニモニック〕
ＴＲＡＰＡ　　ｖｅｃｔｏｒ
〔命令の機能〕
ＴＲＡＰ　ａｌｗａｙｓ
ソフトウエア割り込み、トラップ

・予約命令例外
・P='1' のとき
・cccc='1110,1111' のとき

・条件トラップ命令
〔ニモニック〕
ＲＥＩＴ
〔命令の機能〕
ｒｅｔｕｒｎ　ｆｒｏｍ　ＥＩＴ
ＥＩＴ処理（割り込み、例外処理）からのリターン
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２７２に示す

194

＊〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２６８に示す
〔フラッグ変化〕図２６９に示す
〔解説〕内部割り込み（トラップ）を発生する。この命令は、ユーザプロセスからＯＳを呼ぶ場合などに利用する。ＴＲＡＰＡ命令ではＥＩＴが起動されるので、必ずリング０に入ることになる。ＴＲＡＰ，ＴＲＡＰＡでは、その他のＥＩＴ処理と同じく、ＰＳＳの一部およびＰＳＭのＰＲＮＧのみが更新される。ＰＳＭのＰＲＮＧ以外のフィールド（ＰＳＢを含む）は更新されない。
【０２４０】〔プログラム例外〕
・予約命令例外
・Ｐ＝’１’のとき
・無条件トラップ命令
〔ニモニック〕
ＴＲＡＰ
〔命令の機能〕
ＴＲＡＰ　ｃｏｎｄｉｔｉｏｎａｌｌｙ
条件によるソフトウエア割り込み、トラップ
〔命令オプション〕
／条件指定各種（ｃｃｃｃ）
〔命令ビットパターンとアセンブラ表記〕図２７０に示す
〔フラッグ変化〕図２７１に示す
〔解説〕指定された条件が満たされていた場合に内部割り込み（トラップ）を発生する。ＴＲＡＰ命令ではＥＩＴが起動されるので、必ずリング０に入ることになる。条件の指定方法は、Ｂｃｃ命令と同じである。ＴＲＡＰ，ＴＲＡＰＡでは、その他のＥＩＴ処理と同じく、ＰＳＳの一部およびＰＳＭのＰＲＮＧのみが更新される。ＰＳＭのＰＲＮＧ以外のフィールド（ＰＳＢを含む）は更新されない。ＴＲＡＰで未定義の条件を指定した場合には、予約命令例外（ＲＩＥ）となる。
＊　【０２４１】〔プログラム例外〕

〔フラッグ変化〕図２７３に示す
〔解説〕「本発明装置」では、例外、外部割り込み、内部割り込みを総称してＥＩＴ（Ｅｘｃｅｐｔｉｏｎ，Ｉｎｔｅｒｒｕｐｔ，Ｔｒａｐ）と呼ぶ。ＲＥＩＴ命令は、ＥＩＴからのリターン、すなわち、ＯＳからのリターンや割り込み処理からのリターンを行なうために使用する命令である。この命令は特権命令である。
【０２４２】［ＲＥＩＴのオペレーション］

195　　196

↑TOS ==> PSW;

↑TOS ==> FORMAT/VECTOR;

↑TOS ==> PC;

このほか、EIT のタイプによっては、スタックに付加情報が積まれている場合がある。それをポップしてEIT 発生前の状態に戻す。追加情報の有無は、FORMAT/VECTOR(EITINF) で判定する。また、REIT命令実行時には、DI,DCEのEIT が起動される場合があり、そのチェックを行なう必要がある。詳しくは付録９を参照。

ＦＯＲＭＡＴ／ＶＥＣＴＯＲとして、サポートされないスタックフォーマットが指定されていた場合には、予約スタックフォーマット例外（ＲＳＦＥ）となる。この場合、フォーマットが不当であったスタックフレームは、追加情報の有無が判定できないためにそのまま残し、そのスタックフレームに追加される形でＲＳＦＥのスタックフレームが形成される。この点は、ＲＥＩＴでＤＩ，ＤＣＥを起動する場合とは異なっている。ＤＩ，ＤＣＥの場合には、前のＥＩＴのスタックフレームをクリアしてから、ＤＩ，ＤＣＥの新しいスタックフレームを構成する。

〈〈ＲＳＦＥの処理－ＲＳＦＥで同じスタックを利用する場合〉〉図２７４に示す

ＲＥＩＴ命令で、スタックからポップされたＰＣが奇数であった場合には、奇数アドレスジャンプ例外（ＯＡＪＥ）となる。また、スタックからポップされたＰＳＷによって、ＰＳＷ内のｒｅｓｅｒｖｅｄ（’－’）のビット（ＸＡビットを含む）を’１’に書き換えようとした場合や、ＳＭＲＮＧとしてｒｅｓｅｒｖｅｄの値を書き込もうとした場合には、予約機能例外（ＲＦＥ）となる。ＳＭビットの変化については特にチェックしない。ＥＩＴから戻るためにＲＥＩＴ命令を使う限り、ＲＥＩＴ命令の実行によってＳＭが１から０に変わることはないはずである。しかし、これは運用上の問題として対応することにし、ＲＥＩＴ命令ではＳＭが１から０になったかどうかのチェックは行なわない。

【０２４３】〔プログラム例外〕

・予約命令例外

・Ｐ＝’１’のとき

・特権命令違反例外

・ｒｉｎｇ０以外から実行されたとき

・予約スタックフォーマット例外

・ＥＩＴから復帰する際に、サポートされていないスタックフォーマットが指定されたとき

・奇数アドレスジャンプ例外

・スタックからポップされたＰＣが奇数であったとき

・予約機能例外

・スタツクからポップされたＰＳＷによって、ＰＳＷ内にｒｅｓｅｒｖｅｄの値を書き込もうとしたとき

【０２４４】〔ニモニック〕

ＷＡＩＴ　ｉｍａｓｋ

〔命令の機能〕

ｓｅｔ　ＩＭＡＳＫ　ａｎｄ　ｗａｉｔ

停止、割り込み待ち

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２７５に示す

〔フラッグ変化〕図２７６に示す

〔解説〕ＰＳＷのＩＭＡＳＫフィールドをセットし、プログラムの実行を停止する。外部割り込みまたはリセットにより実行を再開する。この命令は特権命令である。ｉｍａｓｋは、符号なしの数として解釈される。ｉｍａｓｋ≧１６の場合には、予約機能例外（ＲＦＥ）を発生する。外部割り込みの場合には、割り込みが起るまで確定できない情報（ＳＰ１／ＳＰ０のスタックの選択、ベクトル番号）がある。したがって、ＷＡＩＴ命令では、外部割り込みが発生してから情報をスタックに退避する。

【０２４５】

【表１】

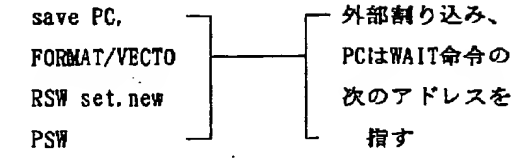
197　　198

[WAITのオペレーション]

imask ==> IMASK

wait for interrupt

<== 外部割り込み



【０２４６】〔プログラム例外〕

・予約命令例外

・ー＝’１’のとき

・特権命令違反例外

・ｒｉｎｇ０から実行されたとき

〔ニモニック〕

ＬＤＣＴＸ　ｃｔｘａｄｄｒ

〔命令の機能〕

ｌｏａｄ　ｃｏｎｔｅｘｔ　ｆｒｏｍ　ＣＴＸＢ　コンテキストのロード

〔命令オプション〕

／ＬＳ　論理空間からＣＴＸＢをロード

／ＣＳ　制御空間からＣＴＸＢをロード　〈〈Ｌ２〉〉

〔命令ビットパターンとアセンブラ表記〕図２７７に示す

〔フラッグ変化〕図２７８に示す

〔解説〕ｃｔｘａｄｄｒで示される実効アドレスをＣＴＸＢＢレジスタにロードし、タスクやプロセスのコンテキストブロック（ＣＴＸＢ）の内容をプロセッサのレジスタにロードする。ロードの行なわれるレジスタは、ＭＭＵの有無やＣＴＸＢＦＭの内容によって変化するが、ＳＰ０～ＳＰ３，ＵＡＴＢ，ＣＳＷなどが含まれる。ＬＤＣＴＸで転送されるレジスタの詳細については、付録８を参照のこと。

【０２４７】／ＬＳオプションを指定した場合には、ｃｔｘａｄｄｒは論理空間のアドレスを意味する。この場合には、論理空間上にＣＴＸＢが置かれていることになる。また、／ＣＳオプションを指定した場合には、ｃｔｘａｄｄｒは制御空間のアドレスを意味する。このオプションは、将来コンテキスト退避用の高速メモリをチップに内蔵した場合に使用することを想定したものであり、現在は〈〈Ｌ２〉〉となっている。これらのオプションは、チップやチップバスのインプリメントに合わせて最も高速なコンテキストスイッチを行なうため、ＣＴＸＢを置く空間に自由度を持たせるという目的で設けられたものである。「本発明装置」では／ＣＳオプションはサポートしない。「本発明装置」標準のMMUを内蔵したプロセッサでは、ＬＤＣＴＸ命令でＵＡＴＢの変更が行なわれる。この場合、ＬＳＩＤを実装しないプロセッサであれば、ＵＡＴＢの変更に伴ってＴＬＢやキャッシュのパージ（ＰＳＴＬＢ／ＡＴに相当する処理）が自動的に行なわれる。ＬＤＣＴＸ命令では論理空間の切り換えが行なわれるため、ＬＤＣＴＸ／ＬＳが意味のある動作を行なうには、ｃｔｘａｄｄｒはＳＲを指している必要がある。ＬＤＣＴＸ／ＬＳでｃｔｘａｄｄｒがＵＲを指していた場合の動作は保証されない。「本発明装置」のＬＤＣＴＸ，ＳＴＣＴＸ命令では、汎用レジスタＲ０～Ｒ１４の転送を行なっていない。これは、次のような理由による。

【０２４８】ー汎用レジスタについては、ＬＤＭ，ＳＴＭ命令で転送することが可能であり、ＬＤＭ，ＳＴＭならばレジスタの指定も可能である。実際のコンテキストスイッチの処理では、入れ換えを行なうレジスタ以外にワーキング用のレジスタが必要になることが多いので、一部のレジスタは転送しない方がよい場合がある。したがって、ＬＤＭ，ＳＴＭのような、より汎用的な命令を使用する方が適当である。

ー現在は、まだコンテキスト退避用のメモリをチップに内蔵することが技術的に難しく、コンテキストの退避には外部メモリを利用せざるを得ない。その場合、ＬＤＣＴＸで汎用レジスタの転送まで行なっても、汎用レジスタの転送を別命令（ＬＤＭ）としても、ほとんど速度差は生じない。

ー将来ＣＴＸＢをすべてチップに内蔵して高速化しようという場合には、ＬＤＣ　ＴＸのｒｅｓｅｒｖｅｄのオプションやＣＴＸＢＦＭの機能を利用して仕様を拡張すれば良い。また、ＬＤＣＴＸ，ＳＴＣＴＸ命令では、ＰＣ，ＰＳＷの転送も行なっていない。これは、次のような理由による。

ー一般に、コンテキストスイッチによって切り換える必要があるのは、ＯＳのＰＣやＰＳＷではなく、ユーザプログラムのＰＣやＰＳＷである。ところが、ユーザプログラムのＰＣやＰＳＷは、普通はＯＳ呼び出し時にスタック中に退避されている。そこで、ＰＣ，ＰＳＷの退避にＳＰ０のスタックを使用するようにしておけば、コンテキストスイッチでＳＰ０を切り換えることにより、ＰＣ，ＰＳＷも間接的に切り替わる。これを積極的に利用

199

し、ＣＴＸＢの構造として、ＳＰ０から間接参照される部分（スタック）にＰＣ，ＰＳＷが置かれるようなものを考えれば、コンテキストスイッチ命令で、ＰＣ，ＰＳＷの操作（スタック～ＣＴＸＢ間のコピー）をする必要がなくなる。

－ＳＰＩを使用した外部割り込みの処理ハンドラの最後で直接コンテキストスイッチを行なう場合には、どうしてもＳＰＩのスタック～ＣＴＸＢ間でＰＣ，ＰＳＷの転送が必要になる。しかし、この場合、外部割り込みの中ではコンテキストスイッチを遅延し、外部割り込みから抜ける時にＤＣＥやＤＩを使ってコンテキストスイッチを行なうようにすれば、ＤＣＥやＤＩでＳＰ０を指定することにより、上記のデータ構造が自然に実現できる。この命令は特権命令である。

【０２４９】ＬＤＣＴＸによってセットされるＰＳＷのｒｅｓｅｒｖｅｄのビット（’－’で表示される）に対して、ＣＴＸＢから’１’をロードしようとした場合には、予約機能例外（ＲＦＥ）が発生する。また、ＵＡＴＢなどの制御レジスタのｒｅｓｅｒｖｅｄのビット（’＝’で表示される）に対して、ＣＴＸＢから’１’をロードしようとした場合には、単に無視される。これは、ＬＤＣによって制御レジスタをセットしようとした場合と同様である。〈〈Ｌ１〉〉仕様のチップでは、ＡＴ＝＊

200

＊００　（アドレス変換なし）の場合にもＵＡＴＢの転送が行なわれる。これは、ＯＳ内のみで一時的にアドレス変換を中止するというケースが考えられるためである。ただし、ＡＴ＝００の場合は、／ＬＳを指定してもｃｔｘａｄｄｒは物理アドレスとして扱われる。ＬＤＣＴＸでＵＡＴＢを転送しないことを指定するには、ＣＴＸＢＦＭを利用する。ＬＤＣＴＸの現在の仕様では、汎用レジスタの転送を行なっていない。しかし、将来仕様が拡張されたり、コンテキスト退避用のメモリをチップに内蔵したりした場合には、ＬＤＣＴＸ命令で複数の汎用レジスタのロードまで行なう予定がある。その場合、ｃｔｘａｄｄｔ／ＥａＡ！Ａで付加モードを許していると、ＬＤＭと同様に、命令が途中で中断した場合の命令再実行が難しくなる。したがって、ＬＤＣＴＸのｃｔｘａｄｄｒ／ＥａＡ！Ａでは付加モードを禁止している。どうしても付加モードの機能を利用したい場合には、ＭＯＶＡを使って

```
MOVA    @(@(@(...))):A. R0
LDCTX   @R0
```

とすることにより、同等の機能が実現できる。

【０２５０】〔プログラム例外〕

・予約命令例外

・XX=’01’～’11’のとき

・EaA がRn, #imm_data,@SP+,@-SP, 付加モードのとき

・特権命令違反例外
・ｒｉｎｇ０以外から実行されたとき
・予約機能例外
・ＰＳＷにｒｅｓｅｒｖｅｄの値を書き込んだとき

〔ニモニック〕

ＳＴＣＴＸ

〔命令の機能〕

ｓｔｏｒｅ　ｃｏｎｔｅｘｔ　ｔｏ　ＣＴＸＢ

コンテキストのストア

〔命令オプション〕

／ＬＳ　論理空間にＣＴＸＢをストア
／ＣＳ　制御空間にＣＴＸＢをストア　〈〈Ｌ２〉〉（検討中）

〔命令ビットパターンとアセンブラ表記〕図２７９に示す

〔フラッグ変化〕図２８０に示す

【０２５１】〔解説〕プロセッサ中の現在のコンテキストの内容を、ＣＴＸＢＢレジスタで示される領域（ＣＴＸＢ）に退避する。退避の行なわれるレジスタは、ＭＭＵの有無やＣＴＸＢＦＭの内容によって変化するが、ＳＰ０～ＳＰ３，ＵＡＴＢ，ＣＳＷなどが含まれる。ＳＴＣＴＸで転送されるレジスタの詳細については、付録８を参照のこと。ＳＴＣＴＸでは、ＬＤＣＴＸと同様に、汎用レジスタやＰＣ，ＰＳＷの転送は行なわれない。ＣＴＸＢＢの指す空間は／ＬＳ，／ＣＳのオプションで指定する。ただし、／ＣＳオプションは、将来コンテキスト退避用のメモリをチップに内蔵した場合にはじめて意味を持つものであり、〈〈Ｌ２〉〉となっている。「本発明装置」では、／ＣＳオプションはサポートしない。「本発明装置」標準のＭＭＵを内蔵したプロセッサでは、ＳＴＣＴＸ命令でＵＡＴＢ退避が行なわれる。この場合、ＳＴＣＴＸ／ＬＳが意味のある動作を行なうには、ＣＴＸＢＢがＳＲを指している必要があるが、ＣＴＸＢＢがＳＲを指すかＵＲを指すかのチェックは特に行なわない。この命令は特権命令である。ＳＴＣＴＸでＣＴＸＢに退避される制御レジスタのｒｅｓｅｒｖｅｄのビットのうち、’－’，’＋’のビットについては、’０’，’１’がＣＴＸＢにセットされる、また、’＝’，’＃’，’＊’のビットについては、ＣＴＸＢにセットされる値は不定であり、インプリメント依存である。これらは、ＳＴＣ命令と同様である。〈〈Ｌ１〉〉仕様のチップでは、ＡＴ＝００（アドレス変換なし）の場合にもＵＡＴＢの転送が行なわれる。これは、ＯＳ内のみで一時的にアドレス変換を中止するというケースが考えられるためである。ただし、ＡＴ＝００の場合は、／ＬＳを指定してもＣＴＸＢＢは物理アドレスとして扱

201

われる。ＳＴＣＴＸでＵＡＴＢを転送しないことを指定するには、ＣＴＸＢＦＭを利用する。

【０２５２】〔プログラム例外〕

・予約命令例外

・ＸＸ＝’００’以外のとき

・Ｐ＝’１’のとき

・特権命令違反例外

・リング０以外から実行されたとき

【０２５３】１２－１６．ＭＭＵ関連命令

〔ニモニック〕

ＡＣＳ　　ｃｈｋａｄｄｒ

〔命令の機能〕

ｔｅｓｔ　ａｃｃｅｓｓ　ｒｉｇｈｔｓ

アクセス権のチェック

〔命令オプション〕なし

〔命令ビットパターンとアセンブラ表記〕図２８１に示す

〔フラッグ変化〕図２８２に示す

〔解説〕ｃｈｋａｄｄｒで指定したアドレスを含むページのＡＴＥを調べ、ｃｈｋａｄｄｒがＰＲＮＧからアクセス可能かどうかをチェックする。チェックした結果に応じてフラッグをセットする。

```
read 可能      ==> M__flag
write 可能     ==> Z__flag
execute 可能   ==> L__flag
```

この命令は特権命令ではなく、ユーザが使用することも可能である。例えばｒｉｎｇ３からＰＲＮＧ＝ｒｉｎｇ３のアクセス権をチェックすることもできる。したがって、ページアウトなどのＯＳの管理すべき情報はできるだけ見せないようになっている。ＡＣＳの実行のために必要なＳｅｃｔｉｏｎ　ｔａｂｌｅやＰａｇｅ　ｔａｂｌｅがページアウトされていた場合には、通常の命令と同じように、ページ不在例外（ＰＯＥ）として命令を再実行する。このほか、ＡＣＳ命令でＡＴＥを参照中には、アドレス変換例外（ＡＴＲＥ）、バスアクセス例外（ＢＡＥ）の発生する可能性がある。ＡＣＳ命令でテストするオペランドのサイズは、バイトと考える。つまり、ＥａＡで示されるアドレスの１バイトが、ＰＲＮＧからアクセス可能かどうかを示す。複数バイトにまたがる領域のチェックを行なう場合は、ソフトウエアで対処する。

【０２５４】ＡＣＳでは、直前のリングからの処理要求に対するアクセス権のチェックを行なう場合に、ＰＲＮＧがそのまま利用できる。しかし、例えばｒｉｎｇ３からｒｉｎｇ２に処理を依頼し、ｒｉｎｇ２からさらにｒｉｎｇ１を呼びたいという場合に、ｒｉｎｇ１でｒｉｎｇ３からのアクセス権をチェックしたい場合がある。この時、ＰＲＮＧはｒｉｎｇ２となっているので、そのままＡＣＳ命令を使うことはできず、ＰＲＮＧをｒｉｎｇ

202

３に書き換えてからＡＣＳを実行する必要がある。このような要求に答えるために、ＰＲＮＧはユーザからも操作可能なＰＳＭに置いている。ＰＲＮＧは、ＡＣＳ命令に対するパラメータという意味を持ったフィールドである。ただし、このままではｒｉｎｇ３からｒｉｎｇ０の保護情報まで見えることになる。そこで、ＰＲＮＧ＜ＲＮＧの時は無条件に

```
L__flag = M__flag = Z__flag = 0
```

とすることによって、保護情報が見えるのを防ぐ。ＡＣＳにおいて、ｃｈｋａｄｄｒが未使用領域（ページ範囲外）であった場合には、Ｒ不可、Ｗ不可、Ｅ不可と同じように、Ｍ__ｆｌａｇ＝０，Ｚ__ｆｌａｇ＝０，Ｌ__ｆｌａｇ＝０アクセス権なしとして命令を正常終了するものとする。ＥＩＴとはならない。

【０２５５】また、ＡＴ＝００（アドレス変換なし）の場合は、リング保護のチェックが行なわれないので、すべてのアドレスに対してアクセス権をもっていると考える。実際には、バスアクセス例外（ＢＡＥ）が発生するためにアクセスできない領域もあるわけだが、そのチェックは行なわない。これは、システムバスに起因するアクセスエラーとメモリ保護に起因するアクセスエラーはレベルの違ったものであり、ＡＣＳでは後者のみのチェックを行なうと考えるためである。したがって、ＡＴ＝００の場合、ｃｈｋａｄｄｒが得られた後はいずれの例外も発生せず、Ｌ__ｆｌａｇ＝Ｍ__ｆｌａｇ＝Ｚ__ｆｌａｇ＝１（アクセス権あり）として命令を終了するものとする。ＡＣＳ命令は、命令エミュレーションのプログラムで、リング保護レベルのチェックまできちんとエミュレートしたい場合に利用できる命令である。エミュレーションのプログラムは通常ｒｉｎｇ０に置かれるので、エミュレートされる命令とは異なったリングで実行されるのが普通である。つまり、リング保護のレベルに関しては、エミュレートされるプログラムとエミュレーションプログラムは異なった環境になっている。そこで、エミュレートされる命令のオペランドをアクセスする前に、ＡＣＳ命令によって、そのオペランドがエミュレートされる命令と同じリング（ＰＲＮＧ）からアクセスできるかどうかをチェックしてやれば、リング保護に関するエミュレーションまできちんと行なうことが可能になる。ＡＣＳのｃｈｋａｄｄｒの実効アドレス計算において、スタックポインタＳＰの参照があった場合には、ＰＲＮＧではなく現在リングＲＮＧのスタックが参照される。

【０２５６】〔プログラム例外〕

・予約命令例外

・EaA がRn, #imm__data,@SP+,@-SPのとき

〔ニモニック〕

ＭＯＶＰＡ　　ｓｒｃａｄｄｒ，ｄｅｓｔ　（「本発明装置」ではサポートしない）

203

〔命令の機能〕
ｍｏｖｅ　ｐｈｙｓｉｃａｌ　ａｄｄｒｅｓｓ
物理アドレスの転送
〔命令オプション〕なし
〔命令ビットパターンとアセンブラ表記〕図２８３に示す
〔フラッグ変化〕第２８４に示す
【０２５７】〔解説〕ｓｒｃａｄｄｒで指定したオペランドの実効アドレス（論理アドレス）を計算し、それを物理アドレスに変換してからｄｅｓｔに転送する。ｓｒｃａｄｄｒで得られた実効アドレスのアドレス変換の方法は通常の命令とは異なり、ＵＡＴＢレジスタではなくＲ１レジスタをアドレス変換テーブルのベースアドレスとして使用する。これは、ＯＳ等から、現在プログラムが走っている論理空間以外の空間に対する操作も行なえるようにするためである。この命令で、高機能命令と同じように固定番号のレジスタを空間指定に使用したのは、高級言語で直接使用することのない命令なので命令の対称性があまり必要ないこと、ビット割り当てからの制約があること、による。ＭＯＶＰＡ命令において、ｓｒｃａｄｄｒを得てから、それを物理アドレスに変換するまでの過程でページ不在例外、アドレス変換例外などが発生した場合、そのエラーはフラッグに反映し、ＥＩＴは起動しない。これはｓｒｃａｄｄｒのアドレス変換に使用するＳｅｃｔｉｏｎ　ＴａｂｌｅやＰａｇｅ　Ｔａｂｌｅがページアウトされていた場合、また最終段のページ（ページテーブルではない）がページアウトの場合、さらに変換テーブルのエントリ（ＡＴＥ）のフォーマットにエラー（予約ＡＴＥエラー）があった場合などが含まれる。この時、ｄｅｓｔは変化せず、Ｖ＿ｆｌａｇがセットされて命令を終了する。また、ページフォールトがあったかどうかはＦ＿ｆｌａｇで示される。エラーやページフォールトがなく、命令を正常に終了した場合には、Ｖ＿ｆｌａｇはクリアされる。この命令は、基本的にはアドレス演算と考えるので、その他のフラッグは無変化である。ＭＯＶＰＡ命令のフラッグ変化をまとめると、図２８５のようになる。なお、ＳＴＡＴＥでＶ＿ｆｌａｇ＝０，Ｆ＿ｆｌａｇ＝１に相当する場合（次の段のページアウト）は、ＭＯＶＰＡではＶ＿ｆｌａｇ＝１，Ｆ＿ｆｌａｇ＝１のページアウトの場合に吸収されているので、ＳＴＡＴＥとＭＯＶＰＡとではフラッグ変化のパターンが異なっている。ｓｒｃａｄｄｒ，ｄｅ＊

204

＊ｓｔなどの実効アドレスを得るまでの過程でページフォールトが生じた場合には、通常の命令と同じようにページ不在例外（ＰＯＥ）が起動される。この命令は特権命令である。
【０２５８】ｄｅｓｔ／ＥａＷ！Ｓでは、@－ＳＰのモードを禁止している。これは、エラーやページアウトの発生によってＶ＿ｆｌａｇがセットされ、ｄｅｓｔの転送ができない場合に、ｄｅｓｔに@－ＳＰが指定されていると命令動作がまぎらわしくなるためである。ＬＤＡＴＥ，ＳＴＡＴＥ，ＬＤＰ，ＳＴＰ，ＬＤＣ，ＳＴＣ，ＭＯＶＰＡ命令の中の特殊空間を参照するオペランドにおいて、付加モードによりメモリの間接参照が起こった場合には、特殊空間の方ではなく論理空間（ＬＳ）の方を参照する。また、スタックポインタＳＰの参照があった場合には、ＰＲＮＧではなく現在リングＲＮＧのスタックが参照される。特殊空間のアドレスという意味を持つのは、最終的に得られた実効アドレスのみである。ＭＯＶＰＡ，ＬＤＡＴＥ，ＳＴＡＴＥ命令で、対象アドレスのＭＳＢが１の場合（ＳＲを示す場合）は、Ｒ１ではなくＳＡＴＢを使ってアドレス変換を行なう。まとめると、図２８６のようになる。ＭＯＶＰＡ，ＬＤＡＴＥ，ＳＴＡＴＥでは、アドレス変換のベースレジスタを、ＵＡＴＢの代わりにＲ１によって指定している。この時、ＵＡＴＢのｒｅｓｅｒｖｅｄの部分に対応するＲ１のビット（’＝’で表現された２＾４，２＾５のビット）が’１’でなくても、特にチェックは行なわず、単に無視される。これは、インプリメントの負担を考慮したためである。チェックが行なわれていなくても、Ｒ１の２＾４，２＾５のビットには必ず’０’を入れてもらうように、マニュアル等で指導する必要がある。ｓｒｃａｄｄｒの実効アドレスを得てから、Ｒ１を使ってアドレス変換を行ない、その物理アドレスを得るまでの動作は、ＡＴビットには影響されない。つまり、ＡＴ＝００であっても、ＡＴ＝０１の時と同じようにｓｒｃａｄｄｒのアドレス変換が行なわれて物理アドレスが得られる。これは、アドレス変換の前準備としてこの命令を使用することを想定したためである。もちろん、ｓｒｃａｄｄｒ，ｄｅｓｔの実効アドレス計算（間接参照など）やｄｅｓｔへの書き込みは、ＡＴ＝００の時物理アドレス対象で行なわれる。
【０２５９】〔プログラム例外〕

・予約命令例外
・+='0' のとき
・W='1' のとき
・EaA がRn, #imm_data, @SP+, @-SPのとき
・EaW!S が#imm_data, @SP+, @-SPのとき
・〈〈Ｌ１〉〉機能例外　　とき
・ＭＯＶＰＡの正しいビットパターンがデコードされた
〔ニモニック〕

205

ＬＤＡＴＥ　ｓｒｃ，ｄｅｓｔａｄｄｒ（「本発明装置」ではサポートしない）

〔命令の機能〕

ｌｏａｄ　ａｄｄｒｅｓｓ　ｔｒａｎｓｌａｔｉｏｎ　＊

/AS　すべての論理空間のTLB をパージする

/SS　R0で指定されたLSIDを持つ論理空間のTLB をパージする

/PT　PTE操作

/ST　STE操作

〔命令ビットパターンとアセンブラ表記〕図２８７に示す

〔フラッグ変化〕図２８８に示す

【０２６０】〔解説〕ｄｅｓｔａｄｄｒで指定したオペランドの実効アドレス（論理アドレス）を計算し、それを物理アドレスに変換する際に使用するアドレス変換テーブルのエントリ（ＡＴＥ）に対して、ｓｒｃで得られるデータの転送を行なう。ｄｅｓｔａｄｄｒに対するアドレス変換の方法は通常の命令とは異なり、ＵＡＴＢレジスタではなくＲ１レジスタをアドレス変換テーブルのベースアドレス（物理アドレス）として使用する。これは、ＯＳ等から、現在プログラムが走っている論理空間以外の空間に対する操作も行なえるようにするためである。ｄｅｓｔａｄｄｒのＭＳＢが１の場合（ＳＲを示す場合）は、Ｒ１ではなくＳＡＴＢを使ってアドレス変換を行なう。／ＰＴオプションの場合、／ＳＴオプションの場合とも、Ｒ１はＳｅｃｔｉｏｎＴａｂｌｅのベースアドレスを指す。結果的に、／ＰＴの場合には２段の間接参照が、／ＳＴの場合には１段の間接参照が行なわれる。ＡＴＥへのセットが正常に行なわれた場合には、ＡＴＥ値の変更によって影響を受けるＴＬＢと論理キャッシュのパージが自動的に行なわれる。

【０２６１】ＬＳＩＤとは、複数のコンテキスト（プロセスやタスク）のＴＬＢの混在を許す場合に、それを区別する番号である。複数の論理空間の区別が可能なＴＬＢの場合には、／ＳＳオプションを指定することによって、ＴＬＢ中のＬＳＩＤとＲ０で示されるＬＳＩＤが一致したＴＬＢのみをパージすることができる。なお、現在使用中の論理空間に対するＬＳＩＤは、ＬＳＩＤ制御レジスタに置かれているが、この命令の実行とは直接関係しない。メモリ管理やＴＬＢの構成はインプリメント依存性の強いところなので、この命令を実装する場合にも、必ずしも／ＳＳオプションをインプリメントする必要はない。また、ＬＳＩＤの機能も必須のものではない。／ＳＳのオプションが用意されているのは、ＬＳＩＤのあるプロセッサとＬＳＩＤのないプロセッサの互換性を取るためである。ＰＳＴＬＢの項を参照のこと。この命令で、高機能命令と同じように固定番号のレジスタを空間指定に使用したのは、高級言語で直接使用するこ

206

＊ｔａｂｌｅ　ｅｎｔｒｙ

ＡＴＥのロード

〔命令オプション〕

<<L2>>

とのない命令なので命令の対称性があまり必要ないこと、ビット割り当てからの制約があること、による。この命令では、ＡＴＥ自体のエラーやページアウトなど、いろいろな場合を見分けるため、Ｆ＿ｆｌａｇ，Ｖ＿ｆｌａｇを使用する。それぞれの場合における動作は、次のようになる。

【０２６２】１．ｄｅｓｔａｄｄｒのアドレス変換に使用するＳｅｃｔｉｏｎ　ＴａｂｌｅやＰａｇｅ　Ｔａｂｌｅのうち、操作する段より上位段のＡＴＥにフォーマットエラー（予約　ＡＴＥエラー）があった場合

この場合は、操作対象となるＡＴＥまで到達できないので、ＡＴＥへのセットは行なわれない。Ｖ＿ｆｌａｇ＝１，Ｆ＿ｆｌａｇ＝０となって命令を終了する。

２．ｄｅｓｔａｄｄｒのアドレス変換に使用するＳｅｃｔｉｏｎ　ＴａｂｌｅやＰａｇｅ　Ｔａｂｌｅのうち、操作する段のＡＴＥを含むテーブル、あるいはそれより上位段のテーブルがページアウトされていた場合

この場合も、操作対象となるＡＴＥまで到達できないので、ＡＴＥへのセットは行なわれない。Ｖ＿ｆｌａｇ＝１，Ｆ＿ｆｌａｇ＝１となって命令を終了する。なお、途中段のＡＴＥで、予約ＡＴＥエラーと次段のページアウトが同時に起こった場合には、予約ＡＴＥエラーの方を優先し、Ｖ＿ｆｌａｇ＝１，Ｆ＿ｆｌａｇ＝０とする。

３．それ以外の場合

この場合には、ｓｒｃのデータがＡＴＥにセットされ、Ｖ＿ｆｌａｇは０となる。ＬＤＡＴＥによってＡＴＥにセットしたデータのＰＩビットが０の場合には、それより下位の段のページアウトを示すため、Ｆ＿ｆｌａｇ＝１となる。また、セットしたデータがＡＴＥとして予約ＡＴＥエラーを起こすものであった場合には、やはりＦ＿ｆｌａｇ＝１となる。この二つのケースは、セットしたＡＴＥを使ってアドレス変換を行なおうとすると例外が発生するという点で共通している。セットしたＡＴＥにエラーがなく、ＰＩビットが１の場合には、Ｆ＿ｆｌａｇ＝０となる。ＬＤＡＴＥ命令のフラッグ変化をまとめると図２８９のようになる。

【０２６３】この命令は、基本的にはアドレス演算と考えるので、Ｍ＿ｆｌａｇ，Ｚ＿ｆｌａｇなどは変化しな

207

い。また、ｓｒｃ，ｄｅｓｔａｄｄｒなどの実効アドレスを得るまでの過程でページフォールトが生じた場合には、通常の命令と同じようにページ不在例外（ＰＯＥ）が起動される。この命令は特権命令である。ＬＤＡＴＥ／ＳＴではＰＳＴＬＢ／ＳＴに相当する処理が、ＬＤＡＴＥ／ＰＴではＰＳＴＬＢ／ＰＴに相当する処理が自動的に行なわれる。ＬＤＡＴＥ，ＳＴＡＴＥ，ＬＤＰ，ＳＴＰ，ＬＤＣ，ＳＴＣ，ＭＯＶＰＡ命令の中の特殊空間を参照するオペランドにおいて、付加モードによりメモリの間接参照が起こった場合には、特殊空間の方ではなく論理空間（ＬＳ）の方を参照する。また、スタックポインタＳＰの参照があった場合には、ＰＲＮＧではなく現在リングＲＮＧのスタックが参照される。特殊空間のアドレスという意味を持つのは、最終的に得られた実効アドレスのみである。ＭＯＶＰＡ，ＬＤＡＴＥ，ＳＴＡＴＥでは、アドレス変換のベースレジスタを、ＵＡＴＢの代わりにＲ１によって指定している。この時、ＵＡＴＢのｒｅｓｅｒｖｅｄの部分に対応するＲ１のビット（'＝'で表現された２＾４，２＾５のビット）が'１'でなくても、特にチェックは行なわず、単に無視さ

208

＊れる。これは、インプリメントの負担を考慮したためである。チェックが行なわれていなくても、Ｒ１の２＾４，２＾５のビットには必ず'０'を入れてもらうように、マニュアル等で指導する必要がある。ＡＴ＝００でＬＤＡＴＥを実行した場合、ｓｒｃのフェッチとｄｅｓｔａｄｄｒの実効アドレス計算は、他の命令と同様にアドレス変換なしで行なわれる。しかし、ＬＤＡＴＥの命令動作そのものは、ＡＴの値に関係しない。すなわち、ＡＴ＝００であっても、得られたｄｅｓｔａｄｄｒの実行アドレスは論理アドレスであると解釈され、それを物理アドレスに変換する際に使用するＡＴＥに対して、ｓｒｃの転送を行なう。これは、アドレス変換の前準備としてこの命令を使用することを想定したためである。ＡＴ＝００の場合のＬＤＡＴＥ，ＳＴＡＴＥ，ＭＯＶＰＡの仕様は、ＡＴ＝０１の場合の仕様との整合性のほか、ＯＳが最初にMMUの動作環境を設定するために利用できるように、また、ユーザプログラムがＡＴ＝０１、ＯＳがＡＴ＝００で動く場合に矛盾なく利用できるように、という意図で決められたものである。

【０２６４】〔プログラム例外〕

・予約命令例外

- !R='11' のとき（!='0' のときは検出しない）
- P='1' のとき
- ttt='010' 〜'111' のとき
- BaR が@-SPのとき
- BaA がRn, #imm__data,@SP+,@-SPのとき

・〈〈Ｌ１〉〉機能例外

・ＬＤＡＴＥの正しいビットパターンがデコードされたとき

〔ニモニック〕

ＳＴＡＴＥ　ｓｒｃａｄｄｒ，ｄｅｓｔ

「本発明装置」ではサポートしない）

〔命令の機能〕

ｓｔｏｒｅ　ａｄｄｒｅｓｓ　ｔｒａｎｓｌａｔｉｏｎ　ｔａｂｌｅ　ｅｎｔｒｙ

ＡＴＥのストア

〔命令オプション〕

／ＰＴ　ＰＴＥ操作

／ＳＴ　ＳＴＥ操作

〔命令ビットパターンとアセンブラ表記〕図２９０に示す

〔フラッグ変化〕図２９１に示す

【０２６５】〔解説〕ｓｒｃａｄｄｒで指定したオペランドの実効アドレス（論理アドレス）を計算し、それを物理アドレスに変換する際に使用するアドレス変換テーブルのエントリ（ＡＴＥ）を読みだし、ｄｅｓｔに設定する。ｓｒｃａｄｄｒに対する実効アドレスのアドレス変換の方法は通常の命令とは異なり、ＵＡＴＢレジスタではなくＲ１レジスタをアドレス変換テーブルのベースアドレス（物理アドレス）として使用する。これは、ＯＳ等から、現在プログラムが走っている論理空間以外の空間に対する操作も行なえるようにするためである。なお、ｓｒｃａｄｄｒのＭＳＢが１の場合（ＳＲを示す場合）は、Ｒ１ではなくＳＡＴＢを使ってアドレス変換を行なう。／ＰＴオプションの場合、／ＳＴオプションの場合とも、Ｒ１はＳｅｃｔｉｏｎＴａｂｌｅのベースアドレスを指す。結果的に、／ＰＴの場合には２段の間接参照が、／ＳＴの場合には１段の間接参照が行なわれる。この命令で、高機能命令と同じように固定番号のレジスタを空間指定に使用したのは、高級言語で直接使用することのない命令なので命令の対称性があまり必要ないこと、ビット割り当てからの制約があること、による。この命令では、ＡＴＥの予約ＡＴＥエラーやページ不在エラーなど、いろいろな場合を見分けるため、Ｆ__ｆｌａｇ，Ｖ__ｆｌａｇを使用する。それぞれの場合における動作は、次のようになる。

【０２６６】１．ｓｒｃａｄｄｒのアドレス変換に使用するＳｅｃｔｉｏｎ　ＴａｂｌｅやＰａｇｅ　Ｔａｂｌｅのうち、操作する段より上位段のＡＴＥに予約ＡＴＥエラーがあった場合

この場合は、操作対象となるＡＴＥまで到達できないので、ＡＴＥの読みだしは行なわれない。Ｖ__ｆｌａｇ＝

209

１，Ｆ＿ｆｌａｇ＝０となって命令を終了する。
２．ｓｒｃａｄｄｒのアドレス変換に使用するＳｅｃｔｉｏｎＴａｂｌｅやＰａｇｅ　Ｔａｂｌｅのうち、操作する段のＡＴＥを含むテーブルあるいはそれより上位段のテーブルがページアウトされていた場合
この場合も、操作対象となるＡＴＥまで到達できないので、ＡＴＥの読みだしは行なわれない。Ｖ＿ｆｌａｇ＝１，Ｆ＿ｆｌａｇ＝１となって命令を終了する。なお、途中段のＡＴＥで、予約ＡＴＥエラーと次段のページアウトが同時に起こった場合には、予約ＡＴＥエラーの方を優先し、Ｖ＿ｆｌａｇ＝１，Ｆ＿ｆｌａｇ＝０とする。
３．それ以外の場合
この場合には、ＡＴＥが読み出されてｄｅｓｔにセットされ、Ｖ＿ｆｌａｇは０となる。ＳＴＡＴＥによって読み出されたＡＴＥのＰＩビットが０の場合には、それより下位の段のページアウトを示すため、Ｆ＿ｆｌａｇ＝１となる。また、読み出されたＡＴＥが予約ＡＴＥエラーを起こすものであった場合には、やはりＦ＿ｆｌａｇ＝１となる。この二つのケースは、読み出されたＡＴＥを使ってアドレス変換を行なおうとすると例外が発生するという点で共通している。読み出されたＡＴＥに予約ＡＴＥエラーがなく、ＰＩビットが１の場合には、Ｆ＿ｆｌａｇ＝０となる。ＳＴＡＴＥ命令のフラッグ変化をまとめると、図２９２のようになる。なお、フラッグ変化の意味を考えると、ＳＴＡＴＥのＦ＿ｆｌａｇ．ｏｒ．Ｖ＿ｆｌａｇに相当するものがＭＯＶＰＡのＶ＿ｆｌａｇとなっており、ＳＴＡＴＥとＭＯＶＰＡとではフラッグ変化のパターンが異なっている。この命令は、基本的にはアドレス演算と考えるので、Ｍ＿ｆｌａｇ，Ｚ＿ｆｌａｇなどは変化しない。また、ｓｒｃａｄｄｒ，ｄｅｓｔなどの実効アドレスを得るまでの過程でページフォールトが生じた場合には、通常の命令と同じようにページ不在例外（ＰＯＥ）が起動される。この命令は特権命令である。
【０２６７】ｄｅｓｔ／ＥａＷ！Ｓでは、@－ＳＰのモードを禁止している。これは、途中段の予約ＡＴＥエラーやページアウトの発生によってＶ＿ｆｌａｇがセットされ、ｄｅｓｔの転送ができない場合にｄｅｓｔに@－ＳＰが指定されていると命令動作がまぎらわしくなるためである。ＬＤＡＴＥ，ＳＴＡＴＥ，ＬＤＰ，ＳＴＰ，ＬＤＣ，ＳＴＣ，ＭＯＶＰＡ命令の中の特殊空間を参照するオペランドにおいて、付加モードによりメモリの間＊

210

＊接参照が起こった場合には、特殊空間の方ではなく論理空間（ＬＳ）の方を参照する。また、スタックポインタＳＰの参照があった場合には、ＰＲＮＧではなく現在リングＲＮＧのスタックが参照される。特殊空間のアドレスという意味を持つのは、最終的に得られた実効アドレスのみである。ＡＴ＝００でＳＴＡＴＥを実行した場合、ｓｒｃａｄｄｒとｄｅｓｔの実効アドレス計算は、他の命令と同様にアドレス変換なしで行なわれる。しかし、ＳＴＡＴＥの命令動作そのものは、ＡＴの値に関係しない。すなわち、ＡＴ＝００であっても、得られたｓｒｃａｄｄｒの実効アドレスは論理アドレスであると解釈され、それを物理アドレスに変換する際に使用するＡＴＥをｄｅｓｔに転送する。これは、アドレス変換の前準備としてこの命令を使用することを想定したためである。ＭＯＶＰＡ，ＬＤＡＴＥ，ＳＴＡＴＥでは、アドレス変換のベースレジスタを、ＵＡＴＢの代わりにＲ１によって指定している。この時、ＵＡＴＢの　ｒｅｓｅｒｖｅｄの部分に対応するＲ１のビット（'＝'で表現された２＾４，２＾５のビット）が'１'でなくても、特にチェックは行なわず、単に無視される。これは、インプリメントの負担を考慮したためである。チェックが行なわれていなくても、Ｒ１の２＾４，２＾５のビットには必ず'０'を入れてもらうように、マニュアル等で指導する必要がある。
【０２６８】〔プログラム例外〕
・予約命令例外
・I='0' のとき
・M='1' のとき
・EaA がRn, #imm_data, @SP+, @-SPのとき
・EaW!S が#imm_data, @SP+, @-SPのとき
・〈〈Ｌ１〉〉機能例外
・ＳＴＡＴＥの正しいビットパターンがデコードされたとき
〔ニモニック〕
PTLB　（「本発明装置」ではサポートしない。
つまり《Ｌ２》である。）
〔命令の機能〕
purge TLB.
TLB のパージ
〔命令オプション〕
/AS　すべての論理空間のTLB をパージする
/SS　R0 で指定されたLSIDを持つ論理空間の TLBをパージする

〔命令ビットパターンとアセンブラ表記〕図２９３に示す
〔フラッグ変化〕図２９４に示す
【０２６９】〔解説〕ＴＬＢのパージを行なう。ＴＬＢのロックやイネーブルなどの細かい操作を行なうには、制御レジスタを用いる。しかし、ＴＬＢに対するパージ操作しか行なわない場合には、そのためだけに制御レジスタを追加するのはインプリメントの負担が大きいた

211

め、ＴＬＢのパージ命令を別に用意している。ＬＳＩＤとは、複数のコンテキスト（プロセスやタスク）のＴＬＢの混在を許す場合に、それを区別する番号である。／ＳＳオプションの際には、Ｒ０により示されたＬＳＩＤを持つ論理空間のＴＬＢのみがパージされる。なお、現在使用中の論理空間に対するＬＳＩＤは、ＬＳＩＤ制御レジスタに置かれているが、この命令の実行とは直接関係しない。ＰＴＬＢ命令では、特定の論理アドレスのＴＬＢのみをパージする機能はなく、指定した論理空間のすべてのＴＬＢがパージされる。特定の論理アドレスのＴＬＢをパージする場合は、ＰＳＴＬＢ命令を使う。ただし、／ＳＳオプションが指定された場合には、指定した論理空間のＵＲのＴＬＢのみがパージされ、ＳＲのパージは一切行なわれない。ＳＲ部分のパージを行なう場合には、必ず／ＡＳを使用する必要がある。この命令は特権命令である。メモリ管理やＴＬＢの構成はインプリメント依存性の強いところなので、この命令は〈〈Ｌ　＊２〉〉となっている。また、この命令を実装する場合にも、必ずしもすべてのオプションをインプリメントする必要はない。ＬＳＩＤの機能も必須のものではない。ＰＴＬＢでは、ＡＴ＝００の場合にも、ＡＴ＝０１の時と同様にパージが実行される。これは、アドレス変換の前準備としてＰＴＬＢ命令を使用することを想定したためである。

212

【０２７０】〔プログラム例外〕

・予約命令例外

〔ニモニック〕

ＰＳＴＬＢ　　ｐｒｇａｄｄｒ　（「本発明装置」ではサポートしない。つまり《Ｌ２》である。）

〔命令の機能〕

ｐｕｒｇｅ　ｓｐｅｃｉｆｉｃ　ＴＬＢ

特定のアドレスのＴＬＢのパージ

〔命令オプション〕

/AS　すべての論理空間のTLB をパージする

/SS　R0 で指定されたLSIDを持つ論理空間のTLB をパージする

/PT　論理アドレス全体（２＾31～２＾12ビット）がprgaddr と一致するエントリをパージする
つまり、PTE の変更時に影響を受ける部分をパージする

/ST　論理アドレスの２＾31～２＾22ビットがprgaddr と一致するエントリをパージする
つまり、STE の変更時に影響を受ける部分をパージする

/AT　論理アドレスの２＾31ビットがprgaddr と一致するエントリをパージする
つまり、UATB.SATB の変更時に影響を受ける部分をパージする

〔命令ビットパターンとアセンブラ表記〕図２９５に示す

〔フラッグ変化〕図２９６に示す

【０２７１】〔解説〕特定の論理アドレスのＴＬＢをパージする。／ＰＴオプションを指定した場合には、対象となる論理空間のＴＬＢのうち、ＳＴＥ～ＰＴＥのインデクスに相当する論理アドレス（すなわち論理アドレス全体）がｐｒｇａｄｄｒと一致するものをパージする。また、／ＳＴオプションを指定した場合には、対象となる論理空間のＴＬＢのうち、ＳＴＥのインデクスに相当する論理アドレスがｐｒｇａｄｄｒと一致するものをパージする。／ＡＴオプションを指定した場合には、対象となる論理空間のキャッシュのうち、論理アドレスのＭＳＢがｐｒｇａｄｄｒと一致するエントリをすべてパージする。ＬＳＩＤとは、複数のコンテキスト（プロセスやタスク）のＴＬＢの混在を許す場合に、それを区別する番号である。／ＳＳオプションの際には、Ｒ０により示されたＬＳＩＤを持つ論理空間のＵＲのＴＬＢのみがパージされる。なお、現在使用中の論理空間に対するＬＳＩＤは、ＬＳＩＤ制御レジスタに置かれているが、この命令の実行とは直接関係しない。この命令は特権命令である。メモリ管理やＴＬＢの構成はインプリメント依存性の強いところなので、この命令は〈〈Ｌ２〉〉となっている。また、この命令を実装する場合にも、必ずしもすべてのオプションをインプリメントする必要はない。ＬＳＩＤの機能も必須のものではない。

【０２７２】／ＡＳ，／ＳＳオプションは、ＬＳＩＤの有無に対する互換性を保つために設けてあるオプションである。意味的には、ＰＳＴＬＢの場合に常に／ＳＳのみが指定できればよいが、ＰＳＴＬＢの場合に常に／ＳＳ指定とすると、ＬＳＩＤの有無によって互換性が失われる恐れがある。例えば、最初にＬＳＩＤの機能のないプロセッサができると、その上で動くプログラムは、Ｒ０にＬＳＩＤのセットを行なわずにＰＳＴＬＢ命令を実行するものになるだろう。同じプログラムを将来ＬＳＩＤの機能の有るプロセッサで実行した場合、その時にＲ

213

０に残っていたゴミによって、全くでたらめのＬＳＩＤに対してＰＳＴＬＢが実行されることになる。これを防ぐためには、オプションを使って、Ｒ０をセットしていない場合には／ＡＳ指定、将来Ｒ０をセットした場合は／ＳＳ指定とすればよいわけで、ＰＳＴＬＢにおける／＊

/SS/PT

/SS/ST

はすべて有効であり、

/SS はR0によって指定される論理空間のURのTLB をパージ

/AS はすべての論理空間に対するTLB のパージ、あるいはLSIDの機能の

は使用しない）

となる。

214

＊ＡＳ指定はこのような意味を持っている。したがって、ＰＳＴＬＢでは、

/AS/PT

/AS/ST

／ＡＳオプションを使えば、ＬＳＩＤのあるプロセッサでもＬＳＩＤのないプロセッサでも共通のプログラムを書けるが、ＬＳＩＤの機能は生かせないことになる。一方、／ＳＳオプションを使えば、ＬＳＩＤの機能は生かせるが、ＬＳＩＤのないプロセッサでは未実装オプションということでエラー（予約命令例外など）になる。

【０２７３】ＰＴＬＢ，ＰＳＴＬＢ命令で／ＳＳオプシ※ョンが指定された場合には、指定した論理空間のＵＲのＴＬＢのみがパージされ、ＳＲのパージは一切行なわれない。ＳＲ部分のパージを行なう場合には、必ず／ＡＳを使用する必要がある。ＰＴＬＢ，ＰＳＴＬＢで／ＳＳのオプションを指定した場合の動作をまとめると、以下のようになる。

PTLB/SS　　R0で指定された論理空間のURをパージ

PSTLB/SS/AT @uraddr　　:uraddrはUR

R0で指定された論理空間のURをパージ

PSTLB/SS/AT @sraddr　　:sraddrはSR /SSでSRを指定したので、何もしない

SR 全体をパージする場合はPSTLB/AS/AT @sraddr を利用

PSTLB/SS/PT @uraddr　　:uraddrはUR

R0で指定された論理空間のURの一部をパージ

PSTLB/SS/PT @sraddr　　:sraddrはSR/SS でSRを指定したので、何もしない

SRの一部をパージする場合はPSTLB/AS/PT @sraddr を利用

ＰＳＴＬＢで／ＳＴオプションの実装が難しい場合には、互換性のため機能を縮退してそのまま実行することにし、ＥＩＴとはしない。具体的には、／ＳＴの代わりに／ＡＴ相当の動作を行なうことになる。ＡＴ＝００でＰＳＴＬＢを実行した場合、ｐｒｇａｄｄｒの実効アドレス計算は、他の命令と同様にアドレス変換なしで行なわれる。しかし、ＰＳＴＬＢの命令動作そのものは、ＡＴの値に関係しない。すなわち、ＡＴ＝００であっても、得られたｐｒｇａｄｄｒの実効アドレスは論理アドレスであると解釈され、ＡＴ＝０１の時と同様にパージが実行される。これは、アドレス変換の前準備としてＰＳＴＬＢ命令を使用することを想定したためである。

【０２７４】〔プログラム例外〕

・予約命令例外

【０２７５】付録１．本発明装置命令セットレファレンス

＊：本発明装置ではサポートしない命令

（データ転送命令）

215 216

	MOV	src.dest	データの移動と符号拡張
	MOVU	src.dest	データの移動とゼロ拡張
	PUSH	src	スタックにプッシュ
	POP	dest	スタックからポップ
	STM	reglist.dest	複数レジスタのストア
	LDM	src.reglist	複数レジスタのロード
	MOVA	srcaddr.dest	実効アドレスを得る
	PUSHA	srcaddr	実効アドレスをスタックにプッシュ

（比較・テスト命令）

CMP	src1.src2	比較、符号拡張と比較
CMPU	src1.src2	ゼロ拡張と比較
CHK	bound.index.xreg	配列の範囲のチェック

＊（算術演算命令）

＊

ADD	src.dest	加算、符号拡張と加算
ADDU	src.dest	ゼロ拡張と加算
ADDX	src.dest	キャリーを含めた加算
SUB	src.dest	減算、符号拡張と減算
SUBU	src.dest	ゼロ拡張と減算
SUBX	src.dest	キャリーを含めた減算
MUL	src.dest	乗算
MULU	src.dest	符号なし乗算
MULX	src.dest.tmp	拡張乗算、サイズが大きくなる
DIV	src.dest	除算
DIVU	src.dest	符号なし除算
DIVX	src.dest.tmp	拡張除算、サイズが小さ剰余を出す
REM	src.dest	剰余
REMU	src.dest	符号なし除算による剰余
NEG	dest	補数演算
<<L2>> INDEX	indexsize.subscript.xreg	多次元配列のアドレス計算

【０２７６】（論理演算命令）

AND	src.dest	論理積
OR	src.dest	論理和
XOR	src.dest	排他的論理和
NOT	dest	全ビット反転

40

（シフト命令）

	SHL	count.dest	論理シフト
	SHA	count.dest	算術シフト
	ROT	count.dest	回転
	SHXL	dest	拡張左シフト
	SHXR	dest	拡張右シフト
	RVBY	src.dest	バイト順の逆転
<<L2>>	RVBI	src.dest	ビット順の逆転

（本発明装置ではサポート）

（ビット操作命令）

217

BTST	offset, base	ビットのテスト
BSET	offset, base	ビットのセット
BCLR	offset, base	ビットのクリア
BNOT	offset, base	ビットの反転
BSCH	data, offset	０または１のサーチ

（１ワード内）

218

＊【０２７７】（固定長ビットフィールド命令）

＊

BFEXT	offset, width, base, dest	ビットフィールドの抽出（符号付き）
BFEXTU	offset, width, base, dest	ビットフィールドの抽出（符号なし）
BFINS	src, offset, width, base	ビットフィールドの挿入（符号付き）
BFINSU	src, offset, width, base	ビットフィールドの挿入（符号なし）
BFCMP	src, offset, width, base	ビットフィールドの比較（符号付き）
BFCMPU	src, offset, width, base	ビットフィールドの比較（符号なし）

（任意長ビットフィールド命令）

BVSCH	０または１のサーチ（任意長ビットフィールド）
BVMAP	ビットマップ演算
BVCPY	ビットマップ転送
BVPAT	パターンとビットマップの演算

（１０進演算命令）

＊ ADDDX	src, dest	BCDの加算
＊ SUBDX	src, dest	BCDの減算
＊ PACKss	src, dest	BCDへのパック
＊ UNPKss	src, dest, adj	BCDからのアンパック

30

SMOV	ストリングのコピー
SCMP	ストリングの比較
SSCH	ストリングのサーチ
SSTR	同一データを繰り返し書き込み（ストリングのフィル）

（キュー操作命令）

【０２７８】（ストリング命令）

QINS	entry, queue	ダブルリンクのキューへ挿入
QDEL	queue, dest	ダブルリンクのキューエントリを削除
QSCH		キューのサーチ

（ジャンプ命令）

219 220

BRA	newpc	ジャンプ（PC相対）
Bcc	newpc	条件ジャンプ（PC相対）
BSR	newpc	サブルーチンジャンプ（PC相対）
JMP	newpc	ジャンプ
JSR	newpc	サブルーチンジャンプ
ACB	step, xreg, limit, newpc	インデクス値を増加するループ命令
SCB	step, xreg, limit, newpc	インデクス値を減少するループ命令
ENTER	local, reglist	スタックフレームの形成、高級言語用サブルーチンジャンプ
EXITD	reglist, adjsp	高級言語用サブルーチンリターンとパラメータ解放
RTS		サブルーチンからのリターン
NOP		ノーオペレーション
PIB		命令キャッシュやパイプラインの整合性をとる

【０２７９】（マルチプロセッサ命令）

BSETI	offset, base	ビットのセット（バスをロック）
BCLRI	offset, base	ビットのクリア（バスをロック）
CSI	comp, update, dest	比較とストア（バスをロック）

（制御空間、物理空間操作命令）

【０２８０】（ＯＳ関連命令）

LDC	src, dest	制御空間へのロード
STC	src, dest	制御空間からのストア
LDPSB	src	PSB へのロード
LDPSM	src	PSM へのロード
STPSB	dest	PSB からのストア
STPSM	dest	PSM からのストア
LDP	src, dest	物理空間へのロード
STP	src, dest	物理空間からのストア

30

* JRNG	vector	リング間コール
* RRNG		リング間リターン
TRAPA	vector	ソフトウエア割り込み、トラップ
TRAP		条件によるソフトウエア割り込み、トラップ
REIT		EIT 処理（割り込み、例外処理）からのリターン
WAIT	imask	停止、割り込み待ち
LDCTX	pcbaddr	プロセスコンテキストのロード
STCTX		プロセスコンテキストのロード

（MMU関連命令）

221　　222

ACS	chkaddr	アクセス権のチェック
* MOVPA	srcaddr,dest	物理アドレスの転送
* LDATB	src,destaddr	ATB のロード
* STATB	srcaddr,dest	ATB のストア
<<L2>>* PTLB		TLBのパージ
<<L2>>* PSTLB	prgaddr	特定のアドレスの TLBのパージ

【０２８１】（符号付き１０進演算命令）

<<L2>>* DCADD	src,dest	符号付きBCD の加算
<<L2>>* DCADDU	src,dest	符号なしBCD の加算
<<L2>>* DCSUB	src,dest	符号付きBCD の減算
<<L2>>* DCSUBU	src,dest	符号なしBCD の減算
<<L2>>* DCX	src,dest	キャリーを含めたBCD の加減算
<<L2>>* DCADJ	src,dest	符号付きBCD の補数
<<L2>>* DCADJU	src,dest	符号なしBCD の補数
<<L2>>* DCADJX	src,dest	キャリーを含めたBCD の補数
<<L2>>* DCCMP	src1,src2	符号付きBCD の比較
<<L2>>* DCCMPU	src1,src2	符号なしBCD の比較
<<L2>>* DCCMPX	src1,src2	キャリーを含めたBCD の比較

【０２８２】付録２．本発明装置のアセンブラ表記について

【０２８３】Ａ２－１．概要

この資料は、命令ニモニック、アドレッシングモードのニモニック、などに関する本発明装置での規定を示したものである。ドキュメントの記述の意味を明確にし、本発明装置に対する理解を深めてもらうことを目的としている。

Ａ２－１－１．このドキュメントにおける記述方法

<...>	メタキャラクタを示す。
[A]	Aはあってもなくてもよい。
{A}*	Aの０回以上の繰り返し
{A}+	Aの１回以上の繰り返し
A ::= B | | C	AはB またはC
A ::= BC	AはB とC をつないだもの

Ａ２－１－２．ニモニック決定の方針

①総称ニモニックとフォーマット別ニモニックを設ける。総称ニモニックは各命令に対応したニモニックであり、短縮形、一般形などフォーマットが複数存在する命令でも、総称ニモニックは一つである。これに対して、フォーマット別ニモニックは、短縮形や一般形などの区別をしたい場合のニモニックである。命令フォーマットを表わす文字を決めておき、総称ニモニックから規則的にフォーマット別ニモニックを作る。ユーザがアセンブラのソースプログラムを書いた場合には、通常総称ニモニックを使う。総称ニモニックに対する最適なフォーマットの選択は、原則としてアセンブラが行なう。

②データタイプ指定子に関して統一的な規則を設ける。データタイプ関係で記述を必要とするものは、演算のためのデータタイプ指定、命令全体でのオペランドサイズ指定、およびオペランド毎のサイズ指定である。これらに関して統一的な規則を設ける。

③ニモニックは、原則としてＩＥＥＥ Ｍｉｃｒｏｐｒｏｃｅｓｓｏｒ Ａｓｓｅｍｂｌｙ Ｌａｎｇｕａｇｅ Ｓｔａｎｄａｒｄ（Ｐ６９４）を標準とする。ただし、これには一般的な感覚になじまないと思われるところ、本発明装置のアーキテクチャに合わないところなどがあるので、あくまでも個々の名称を決める際の参考とするだけである。考え方や規則まで完全にＩＥＥＥに合わせるわけではない。

④特殊記号の利用はできるだけ避ける。ここで定義するアセンブラでは、できるだけ特殊記号を使用しないという方針にしている。それは、オペランドに数式が来たり、アセンブラを拡張した場合に、その中で使用する記号と競合させないためである。また、文字セットの少ない大型機でも開発を行なうためには、あまり多くの記号を使用するのは望ましくない。特殊記号の利用をできるだけ避けたため、アセンブラの中では括孤を一種類しか使用しておらず、また’；’，’＆’などが未使用の特殊記号となっている。

【０２８４】Ａ２－１－３．アセンブラ命令

本発明装置用アセンブラ言語における一つの命令は、一つのオペレーションニモニックと複数個（０個，１個を含む）、のオペランドニモニックにより記述される。オペコードニモニックとオペランドニモニックの間は一個

223

以上の空白文字（スペースまたはタブ）により区切られ、オペランドニモニックどおしの間はコンマ'，'に＊

224

＊より区切られる。

〈アセンブラ命令〉::=

〈オペレーション〉[〈オペランド〉{,〈オペランド〉}＊]

Ａ２－１－４．オペランドの順序

オペランドの順序は命令毎に定まっているが、原則は次のようになる。

移動命令（ＭＯＶ）

第一オペランドがソース、第二オペランドがデスティネーションになる。すなわち、

第一オペランド＝＝〉第二オペランド

これはＩＥＥＥ標準と同じである。

２項演算の２オペランド命令（ＳＵＢなど）

第一オペランドが２番目のソース、第二オペランドが１番目のソースとデスティネーションになる。すなわち、

第二オペランド．ｏｐ．第一オペランド＝＝〉第二オペ※ランド

※ランド

これはＩＥＥＥ標準とは異なるが、多くのプロセッサで用いられている方法であり、なじみやすい。

【０２８５】Ａ２－２．オペレーションのニモニック

Ａ２－２－１．ニモニック生成規則

ＩＥＥＥでは、演算操作を示す動詞をニモニックの先頭にもってくるという考え方であるが、本発明装置ではさらにその前にデータタイプ指定子を置く。演算操作そのものに対するニモニックは、ほぼＩＥＥＥに合わせる。本発明装置での命令のニモニックは、次のような規則で生成する。

【０２８６】

〈オペレーション〉::=

[〈データタイプ〉]〈演算操作〉{〈バリエーション〉}＊{/〈オプション〉}＊{:〈フォーマット〉}＊[.〈サイズ〉]

例：

```
MOV
SMOV/NE.W
MOV.W
MOV:L
MOV:Q.W
```

〈データタイプ〉　命令の先頭で指定するのは、演算方法に大きな影響を与えるデータタイプ、すなわち、〈演算操作〉に対して直交関係にないデータタイプである。このデータタイプには、ストリング、キュー、ビットフィールドなどが含まれる。データサイズ（整数の８，１６，３２，６４ビット、浮動小数の３２，６４ビットなど）の指定は、ここではなく〈サイズ〉で行なう。また、符号付き、符号なしの指定、およびアドレス演算の指定は、ここではなく〈バリエーション〉で行なう。

〈演算操作〉　演算そのものの指定を行なう。できる限りＩＥＥＥに合わせる。条件ジャンプ命令の条件の指定は本来オプションとするべきであるが、慣例にしたがって基本部分の〈演算操作〉に含める。

〈バリエーション〉演算に対する細かい操作や属性の指定を行なう。

〈オプション〉　命令フォーマット中の数ビットで表現される命令オプションを示す。オプションになるのは、ストリング命令の終了条件、キューのサーチ条件などである。

【０２８７】〈フォーマット〉　短縮形、一般形などのフォーマットを指定する。通常は書かなくてもよく、書かない場合は総称ニモニックになる。アセンブラのソースで〈フォーマット〉を書かずに総称ニモニックを使った場合には、アセンブラで自動的に最適なフォーマットを選ぶ。〈フォーマット〉を書いた場合にはフォーマット別ニモニックの記述になる。アセンブラのソースでユーザが〈フォーマット〉を書いた場合には、強制的にそのフォーマットを使うことを示す。〈フォーマット〉によるフォーマット別ニモニックは、仕様書やマニュアルの記述において、あるいは逆アセンブラなどにおいて、あえて命令フォーマットの区別をしたい場合に用いる。

〈サイズ〉　オペランドのサイズを指定する。

〈サイズ〉を使用する命令は、主として整数を扱う命令と浮動小数を扱う命令である。〈サイズ〉は〈データタイプ〉とは異なり、〈演算操作〉に対して直交関係があるのが特徴である。

【０２８８】Ａ２－２－２．データタイプ

〈データタイプ〉を表わす文字として、次のようなものがある。なし　整数演算、アドレス演算、雑命令など

225

F	浮動小数
S	ストリング
Q	ダブルリンクによるキュー
B	１ビットのビット操作
BF	固定長ビットフィールドの操作
BV	任意長ビットフィールドの操作

【０２８９】Ａ２－２－３．演算操作
原則としてＩＥＥＥのニモニックに従う。使用するものは次の通りである。ＡＤＤ，ＳＵＢ，ＭＵＬ，ＤＩＶ，ＣＭＰ，ＮＥＧ，ＡＮＤ，ＯＲ，ＸＯＲ，ＮＯＴ，ＬＤ，ＳＴ，ＭＯＶ，ＰＵＳＨ，ＰＯＰ，ＷＡＩＴ，ＮＯＰ
注意
・ＭＯＶ，ＬＤ，ＳＴの使い分け
ＭＯＶ　レジスタ間、メモリ間の転送
ＬＤ　メモリからレジスタへの転送
ＳＴ　レジスタからメモリへの転送
ＬＤ，ＳＴは、方向性を意識する必要のある命令に使用

226

する。
・シフト関係のオペレーションは、左右の指定方法が異なるためにＩＥＥＥのニモニックをそのまま使うわけではないが、ＩＥＥＥの原則を生かしてＳＨＡ，ＳＨＬ，ＲＯＴとする。
・ブランチ（条件分岐）命令に関しては、ＩＥＥＥに従うと’ＢＶ’などが別の意味とぶつかること、符号付き整数の比較と符号なし整数の比較の区別をわかりやすくしたいこと、などを考慮したため、条件指定の部分がＩＥＥＥには従っていない。
・ＪＭＰ，ＪＳＲ，ＲＴＳは、ブランチ命令とのバランスからＩＥＥＥには従っていない。
・〈バリエーション〉の’Ｘ’により拡張演算を表わすことに統一したため、ＡＤＤＸ，ＳＵＢＸ，ＭＵＬＸ，ＤＩＶＸについてもＩＥＥＥに従っていない。
【０２９０】Ａ２－２－４．バリエーション
〈バリエーション〉は、演算に対する属性などを指定するものである。次のような文字を使用する。

X	拡張演算	例:ADDX.MULXなど
U	符号なしデータの演算	例:MOVU.ADDU.MULU など
C	制御空間（制御レジスタ）に対する演算	例:LDC,STC
P	物理空間に対する演算	例:LDP.STP
I	バスをロックして行なう処理	例:BSETI.BCLRI.CSI
M	複数データの処理	例:LDM.STM
A	アドレス計算	例:MOVA.PUSHA.MOVPA
D	10進演算（符号なし、データチェックなし）またはスタック上のパラメータを捨てる処理	例:ADDDX.SUBDX 例:EXITD

【０２９１】Ａ２－２－５．フォーマット
〈フォーマット〉は、命令フォーマットの詳細を区別したい場合に用いる。次のような文字を使用する。

Q	リテラル短縮形 ビットフィールド命令のスタティック形 ループ命令のリテラル短縮形

例：MOV:Q.W　#3,@abs
　　BTST:Q.B　#4,@abs
　　ACB:Q　#1,R1,#5,loop1

227　　228

R　　レジスターレジスタ間演算の短縮形
ループ命令のレジスタ短縮形
例：AND:R.W　　R1,R2
MOVA:R.W　　@(disp:16,R2),R3
ACB:R　　#1,R1,R2,loop2

L　　メモリーレジスタ間演算の短縮形
例：ADD:L.W　　@abs,R2
MOV:L.W　　@(disp,R2),R3

S　　レジスターメモリ間演算の短縮形（MOV のみ）
例：MOV:S.W　　R2,@abs

I　　イミディエート短縮形
例：ADD:I.W　　#100000,@abs2

G　　２オペランド命令の一般形
ループ命令の一般形
例：ADD:G.W　　@abs1,@abs2
ACB:G　　@abs1,R1,@a
bs2,loop3

E　　２オペランド命令の一般形の８ビットイミディエート
例：ADD:E.W　　#100.B,@abs2

8　　newpc が８ビット
例：ACB:G　　@abs1,R1,@abs2,
loop3:8

16　　newpc が16ビット
例：BEQ:G　　error:16

32　　newpc が32ビット
例：BNE:G　　next:32

64　　newpc が64ビット
例：BRA:G　　loop:64

なお、ここで示した’：Q’，’：G’．．．といったフォーマット指定は、一つの命令（総称ニモニック）の中でフォーマットの区別を行ない、フォーマット別ニモニックを作ることが目的のものである。つまり、アセンブラ表記上のフォーマット指定である。一方、命令フォーマット説明で用いたＧ－ｆｏｒｍａｔ，Ｅ－ｆｏｒｍａｔ．．といったフォーマットの方は、命令全体の中でのフォーマットの説明を行なうことが目的のものである。したがって、例えば同じ’：G’であっても’ＭＯＶＡ：G’であればＭＯＶＡ命令の一般形なのでＧＡ－ｆｏｒｍａｔであり、’ＭＯＶ：G’であればＭＯＶ命令の一般形なのでＧ－ｆｏｒｍａｔということになる。

【０２９２】Ａ２－２－６．サイズ

ＩＥＥＥでは６４ビット整数まで考慮されていないので、扱うデータサイズはどうしてもＩＥＥＥと異なったものになる。

整数の場合

４通りのサイズが対称的にサポートされている点、オペランド側でもデータタイプが指定できる点、が特徴であ

229

る。オペレーション側にもオペランド側にも同じものを書くので、'．'により区切っている。〈サイズ〉として、次のような文字を使用する。

B	Byte	8ビット長の整数
H	Halfword	16ビット長の整数
W	Word	32ビット長の整数
L	Longword	64ビット長の整数

'L'は本発明装置32では使用できない。

浮動小数点の場合

別に定める。

【０２９３】Ａ２－３．オペランドのニモニック

オペランドには、汎用のアドレッシングモードまたはそのサブセットが利用できるもの（一般オペランドと呼ぶ）と、命令に応じた特殊な指定をするもの（特殊オペランドと呼ぶ）とがある。特殊オペランドについては、命令毎にフォーマットを定める。特殊オペランドをとる命令は、

BRA. Bcc. BSR. ACB. SCB	〈newpc オペランド〉
LDM. STM	〈reglist オペランド〉

などである。

〈オペランド〉::=
　　〈一般オペランド〉
　|　〈特殊オペランド〉

一般オペランドに関しては、本発明装置において、オペランド毎にデータサイズを指定できる点が特色であり、アセンブラにおける一般オペランドの記述方法にもその＊

230

＊ような能力を持たせている。また、オペランドに対しても総称ニモニックとフォーマット別ニモニックを設けている。一般オペランドのニモニックは、実際のオペランドの値（実効アドレス）と付加モードフォーマットの区別、およびサイズから成る。

〈一般オペランド〉::=
　〈オペランド値〉[:〈付加モード指定〉][.〈サイズ〉]

【０２９４】Ａ２－３－１．アドレッシングモード表記の原則

従来のプロセッサでは、アドレッシングモードの数があまり多くなかったため、それぞれのモードを個別に考え、別々の記号を割り当てておけばよかった。また、表記法と実際のアドレッシングのオペレーションがうまく対応していない場合も見られた。例えば、あるプロセッサではレジスタ相対間接のアドレッシングモードをｄｉｓｐ（Ｒｎ）で表現する場合があるがこれはオペレーションとしてはｍｅｍ［ｄｉｓｐ＋Ｒｎ］であり、ｄｉｓｐの部分とＲｎの部分の扱いが対称的ではない。これだけで使っている場合には問題は生じないが、これを組み合わせて複雑なモードを作っていった場合には矛盾の起きることがある。本発明装置では付加モードといった機能があるため、統一的、規則的なアドレッシングの表記を行なわないと混乱を招く。そこで、本発明装置では実際のオペレーションとその表記との関係について原則を設け、それに基づいて、付加モードまで一貫したアドレッシングモードの表記を行なうこととした。アドレッシングは、基本的には加算と間接参照の繰り返しである。したがって、この二つのオペレーションに対する表記法が決まればよい。本発明装置の表記原則をまとめると次のようになる。

［本発明装置アドレッシングモードの表記原則］

@Aまたは@(A)	アドレスA のメモリ内容を参照 mem[A]
@(A.B.C.,...)	A.B.C.. を加算して、加算結果のアドレスのメモリ内容を参照 mem[A + B + C +...]

本発明装置における'()'は、間接参照などの特別な意味は持たない。一般の数式と同じように、結合の順序を示すに過ぎない。したがって、@Ａと@（Ａ）は全く同じ意味ということになる。以下で説明するシンタックスにおいて'（．．）'が入る場合であっても、'（．．）'内に一つの項しかなければ、'（．．）'を省略しても構わない。

【０２９５】従来のプロセッサでは'（．．）'によって間接参照を意味する場合があり、これがある程度慣用的な表記法となっている。しかし、このような表記法では以下のような点で誤解を生みやすい。

231　　232

例：

慣用的表記法	オペランド値
Rn	Rn
(Rn)	mem[Rn]
abs	mem[abs]またはabs
(abs)	mem[mem[[　abs]]]またはmem[abs]

うまく対応がとれない場合が生ずる。

本発明装置において、間接参照を必ず'@'によって表現しているのは、こういった理由による。イミディエート、スタック操作のアドレッシングモード、インデクスのスケーリングなどの処理はこの原則に入らないので、原則を参考にしながらそれぞれ別に表記法を定める。

【０２９６】Ａ２－３－２．付加モードの指定

<付加モード指定> ::= A | N

'A'の指定は、付加モードのフォーマットを使うということを特に強調したい場合に付け加える。また、'＊N'の指定は、付加モードを使用しないということを特に強調したい場合に付け加える。これらの指定は、フォーマット別ニモニックに相当するものである。':N','：A'とも書かれていない場合は、そのアドレッシングが付加モード以外の短いモードで実現できるかどうかをアセンブラが判断し、実現できればそのモードを使う。付加モードでないと実現できなければ、付加モードを使う。

例：

@(disp.PC):A	常にPC相対付加モードを使用 dispが32ビット以下であっても付加モードを使用
@(disp.PC):N	常にPC相対間接のモードを使用 dispが64ビットであるとエラー
@(disp.PC)	dispが32ビットであればPC相対間接のモードを使用 dispが64ビットであればPC相対付加モードを使用

【０２９７】Ａ２－３－３．サイズ

〈サイズ〉は、オペランドの演算サイズを示すものであり、オペレーションのニモニックに示されたサイズと組みになって実際のサイズ指定を行なう。サイズ指定の文字は、オペレーションに使われるものと同じである。オペランド中の〈サイズ〉と、オペレーション中の〈サイズ〉の関係は、原則として次のようになる。

・オペレーション中に〈サイズ〉の指定があった場合には、その〈サイズ〉が全部のオペランドのデフォルトの※サイズとして有効になる。ただし、サイズの指定ができないオペランド、イミディエートのオペランド、特殊な意味を持つオペランドの場合はこの限りではない。

・オペランド中に〈サイズ〉の指定があった場合には、それがそのオペランドのサイズになる。オペレーション中で異なるサイズが指定されていても、オペランドで指定された〈サイズ〉の方が優先される。

・オペランド中で〈サイズ〉の指定を行ない、それが利用できないサイズであった場合はエラーとなる。

例：

MOV.W	@src.@dest	src.destともW(WORD) 型
MOV.W	@src.B.@dest	src はB(BYTE) 型、destはW(WORD) 型
MOV	@src.B.@dest.W	src はB(BYTE) 型、destはW(WORD) 型

【０２９８】Ａ２－３－４．オペランド値

以下の項では、一般オペランドに関してアドレッシングモード別にそのアセンブラ表記を示す。〈イミディエート値〉、〈絶対アドレス〉などの内容としては、数値、変数名、数式などが書けるが、そのシンタックスは別に定める。〈フォーマット〉は、アドレッシングモードのフォーマット選択を明示したい場合に書く。主としてアドレッシングモードの拡張部のサイズを指定するために使用する。通常は書かなくてもよく、書かない場合はアセンブラで自動的に最適なフォーマット（サイズ）を選ぶ。〈フォーマット〉によるフォーマット別ニモニックは、仕様書やマニュアルの記述において、あるいは逆ア

センブラなどにおいて、あえてアドレッシング部のフォーマットの区別をしたい場合に用いる。

＜フォーマット＞ ::= 4 | | 16 | | 32 | | 64

4　　4 ビット長のアドレッシング修飾部

16　　16ビット長のアドレッシング拡張部
@(disp:16,Rn),abs:16など

32　　32ビット長のアドレッシング拡張部
@(disp:32,Rn),abs:32など

64　　64ビット長のアドレッシング拡張部
abs:64など

【０２９９】〈フォーマット〉により指定されるのは、あくまでも命令フォーマット自体のサイズである。一方、〈サイズ〉により指定されるのは、演算されるオペランドのサイズである。イミディエートモードの場合を除けば、両者は全く異なるものである。

例：

```
MOV     R0.W,@addr:16.W
```

この命令では、Ｒ０の内容を’ａｄｄｒ’で示されるメモリに転送する。絶対アドレッシングが用いられている。’：１６’は’ａｄｄｒ’を１６ビットで表現することを示す。したがって、’ａｄｄｒ’の範囲は＄ｆｆｆｆ８０００～＄００００７ｆｆｆとなる。一方、’.Ｗ’は演算をＷＯＲＤ（３２ビット）で行なうことを示す。つまり、この命令により４バイトのデータが転送される。〈レジスタ番号〉に入るのは、汎用レジスタのニモニックである。

＜レジスタ番号＞ ::=

R0 | | R1 | | R2 | | R3 | | R4 | |
R5 | | R6 | | R7 | | R8 | | R9 | | R10 | |
R11 | | R12 | | R13 | | R14 | | R15 | |
FP | | SP

ＦＰとＲ１４は全く同義、ＳＰとＲ１５は全く同義である。

【０３００】Ａ２－３－４－１．レジスタ直接

オペランド＝Ｒｎ

＜オペランド値＞ ::=
＜レジスタ番号＞

例：

R1

Ａ２－３－４－２．レジスタ間接

オペランド = mem[disp16 + Rn]
mem[disp32 + Rn]

＜オペランド値＞ ::=
@(＜ディスプレースメント＞[:＜フォーマット＞],＜レジスタ番号＞)

＜フォーマット＞ ::= 16 | | 32

例：

@(disp:16,R5)

オペランド = mem[Rn]

＜オペランド値＞ ::=
@＜レジスタ番号＞

例：

@R2

【０３０１】Ａ２－３－４－３．レジスタ相対間接

235

Ａ２－３－４－４．リテラルとイミディエート
オペランド = imm_data

＜オペランド値＞ ::=
　　#＜リテラル値＞

＜オペランド値＞ ::=
　　#＜イミディエード値＞

リテラルの命令フォーマットを使用するということを明
例：

```
ADD:Q.W #1.R0    リテラルのフォーマットを使用（２バイト）
ADD:I.W #1.R0    イミディエート形のフォーマットを使用（６バイト）
                 ソースオペランド'1'は32ビットで表現
ADD:L.W #1.R0    短縮形のフォーマットを使用（６バイト）
                 ソースオペランドとして32ビット
                 イミディエートを指定
ADD:G.W #1.B.R0
                 一般形のフォーマットを使用（６バイト）
                 ソースオペランドとして８ビット
                 イミディエートを指定
                 16ビットフィールドの下位８ビットを用いて'1'を表
                 現。'1'は32ビットに符号拡張される。
ADD:E.W #1.R0
                 一般形８ビットイミディエートのフォーマットを使用
                 （４バイト）'1'は32ビットに符号拡張される。
ADD:G.W #1.R0
                 #1にはサイズの指定がなく、しかも:Gフォーマットな
                 のでサイズの自由度が残されている。したがって、自
                 動的に最小のサイズが選択され、
                     ADD:G #1.B.R0.W （６バイト命令）
                 と等価になる。
                     ADD:G #1.W.R0.W （８バイト命令）
```

236

示したい場合は、オペレーションのニモニック中で示す。イミディエートの場合には、拡張部のサイズがオペランドのサイズとして決まるため、〈フォーマット〉と〈サイズ〉が同じ意味になる。アセンブラとしては、〈フォーマット〉としても〈サイズ〉としてもその大きさを指定することができる。

【０３０２】なお、イミディエートのオペランドでオペランド側にサイズ指定がなく、しかも命令機能の上でサイズの自由度がある場合には、自動的に最小のサイズが選択されるものとする。

237 238

になるのではない。

ADD:G.W #1:16,R0

これは、＜サイズ＞ではなく＜フォーマット＞によって命令を選択した例であり、

ADD:G.W #1.H.R0

と等価になる。

総称ニモニックで単に

ADD.W #1,R0

と書くと、最もコードの短い

ADD:Q.W #1.R0

が選択される。

【０３０３】また、命令によっては、サイズが一つに固定されているわけではないが、実質的にほとんど一つのサイズでのみ使用されるものがある。そのようなものについては、特にオペランド側に〈サイズ〉がついていな＊い限り、命令によって定められるデフォルトサイズを適用する。これは、〈オペレーション〉のニモニックが全体のオペランドにかかるという原則に対しては、例外となる。

[例]

ビット操作命令のアクセスサイズ（BB指定）は、８bit(.B)がデフォルト.H..W は<<L2>>..L は<<LX>>固定長ビットフィールド操作命令のレジスタサイズ（X 指定）は、32bit(.W)がデフォルト、

.H..B は使用不可、.Lは<<LX>>

BTST.W R0, @addr = BTST R0.W, @addr.B

Ａ２－３－４－５．アブソリュート

オペランド = mem[abs16]

mem[abs32]

mem[abs64]

30

＜オペランド値＞ ::=

@＜絶対アドレス＞[:＜フォーマット＞]

＜フォーマット＞ ::= 16 | | 32 | | 64

例：

@abs:32

40

【０３０４】Ａ２－３－４－６．ＰＣ相対間接

239

オペランド = mem[disp16 + PC]
　　mem[disp32 + PC]

＜オペランド値＞ ::=
　　@([＜ディスプレースメント＞[:＜フォーマット＞]],PC)

＜フォーマット＞ ::= 16 | | 32

例：
　@(disp,PC)

Ａ２－３－４－７．スタックポップ

240

オペランド = mem[SP++]

＜オペランド値＞ ::=
　　@SP+

例：
　@SP+

【０３０５】Ａ２－３－４－８．スタックプッシュ

オペランド = mem[--SP]

＜オペランド値＞ ::=
　　@-SP

例：
　@-SP

＊【０３０６】Ａ２－３－４－９．ＦＰ相対間接

＊30

オペランド = mem[disp4 + FP]

＜オペランド値＞ ::=
　　@([＜ディスプレースメント＞[:＜フォーマット＞]],＜レジスタ番号＞)

＜フォーマット＞ ::= 4

＜レジスタ番号＞ ::= FP | | R14

例：
　@(disp4:4,FP)

このアドレッシングモードでは、ビットパターン中に指定されたｄｉｓｐ値を４倍して実際のディスプレースメントとするが、アセンブラでの表記に使う値は、４倍した後のものである。〈フォーマット〉を指定しない場合には、アセンブラでの表記がレジスタ相対間接のモードと同じになるため、アセンブラによって最適なモードが選択される。つまり、@（ｄｉｓｐ，Ｒｎ）と書かれたオペランドでは、ＲｎがＲ１４またはＦＰであり、かつｄｉｓｐが－３２～３１の４の倍数の時にはＦＰ相対間接のモードが選択され、それ以外の場合にはレジスタ相対間接のモードが選択されるわけである。

50

【０３０７】Ａ２－３－４－１０．ＳＰ相対間接

241　　242

オペランド = mem[disp4 + SP]

〈オペランド値〉 ::=
　　@([〈ディスプレースメント〉[:〈フォーマット〉]],〈レジスタ番号〉)

〈フォーマット〉 ::= 4

〈レジスタ番号〉 ::= SP | | R15

例：
　@(disp4:4,SP)

このアドレッシングモードでは、ビットパターン中に指定されたｄｉｓｐ値を４倍して実際のディスプレースメントとするが、アセンブラでの表記に使う値は、４倍した後のものである。

Ａ２－３－５．付加モード

付加モードについても、機能面の要求を示す総称ニモニックと、フォーマットやビットパターンを記号化したフォーマット別ニモニックを設ける。

【０３０８】［総称ニモニックに関して］

・@または@（．．．）により間接参照を表わし、（．．．，．．．，．．．）によりアドレスの加算を表わすという原則はそのままである。

・表記の順序は、

ベースモードまたはそれまでの付加モードtmp 値
==> ディスプレースメント
==> インデクス

を原則とする。こうすると、実効アドレス計算の流れが左から右への単純な形になり、先の段の付加モードに必要な情報が先の方に、後の段の付加モードに必要な情報が後ろの方に集まる。すなわち、総称ニモニックの表記の順序が、付加モードの機械語ビットパターンの順序と同じになる。したがって、フォーマット別ニモニックや※

※実際の機械語の付加モードとの対応がよく、アセンブラも簡単になり、理解しやすくなる。

【０３０９】［フォーマット別ニモニックに関して］

・フォーマット指定用として、つぎの３つの文字を導入することにより、機械語のビットパターンと１対１に対応した表記ができるようにする。

:B　その部分の処理をベースモードにより実現することを表わす
:A　その部分までの処理を一段の付加モードにより実現することを表わす
:N　その部分の処理は次の段の付加モード（':A'の指定のある部分）でまとめて実現することを表わす

なお、「その部分の処理」とは、フォーマット指定文字がディスプレースメントやレジスタに付いている場合にはその値の加算処理を、フォーマット指定文字が閉じ括弧'）'についている場合は間接参照の処理を意味する。また、'：A'で「その部分までの処理」とあるのは、その'：A'の部分の処理と、それより左側で'：N'の付いた部分（前の'：A'または'：B'との間で'：N'の付いた部分）の処理を合わせて行なうことを示す。

・フォーマットをすべて指定した場合には、'：A'の個数が付加モードの段数になる。また、通常は一回の'：A'が一段の間接参照に対応する。ただし、複数のインデクスレジスタを加算する場合（間接参照がなくても'：A'が必要）、最後の段で二重間接を行なう場合（二段の間接参照でも一回の'：A'でよい）は例外である。

・フォーマットの表記のない場合には、総称ニモニックとして表記した処理を実現できるような付加モードが自動的に選択される。また、実際の付加モードでは実現できないフォーマットをフォーマット別ニモニックで指定した場合は、エラーとなる。さらに、フォーマット指定ニモニックからフォーマット指定文字を取り去ると、そのまま総称ニモニックになる。このような点は、フォーマット別ニモニックの一般的な原則と同じである。

【０３１０】［フォーマット一般に関して］

・複数のアドレス加算がない場合、@（．．．）の　括

243

弧は書かなくてもよい。よって、例えば　付加モードで実現される３重間接参照の@（@（@（Ｒ１））は@@@Ｒ１と書いてもよい。これは、付加モード以外のアドレッシングにも適用される原則であり、一種のシンタックスシュガーと言える。

・インデクスのスケール値は、ＩＥＥＥでは’：B’，’：W’などサイズ指定文字を使用しているが、

244

将来スケール値にもっと大きな値を入れることも考えられるので、ここは従来通りスケール値の数字を直接書くようにする。また、スケーリングを指定する文字もＩＥＥＥの’：’ではなく、’＊’を使用する。これは、’：’をフォーマット指定の目的で別に使用しているためである。

例：

@(offset.PC)

mem[offset + PC]'

総称ニモニック。offsetが32ビットに入ればPC相対間接モード

で実現し、32ビットに入らなければ付加モードで実現する。

@(offset.PC):N

mem[offset + PC]

必ずPC相対間接モードで実現し、付加モードにはしない。

本発明装置64で、offsetが32ビットに入らない時はエラーとなる。

@(offset[:N],PC[:N]):A

mem[offset + PC]

必ず付加モードで実現する。ベースモードの指定に相当する部分がないので、

絶対付加モード

＋付加モード　BI=10, disp= offset, index=PC, scale=1

になる。

@(PC[:B],offset[:N])[:A]

mem[offset + PC]

必ず

PC相対付加モード

＋付加モード　BI=10, disp=offset, index=0, scale=*

で実現する。

@(@(@(R3[:B],base1[:N],R4*4[:N])[:A],ba se2[:N],R5[*1][:N])[:N])[:A]

mem[mem[mem[R3 + base1 + R4*4] +base2 + R5]]

R3相対付加モード

＋付加モード　BI=01, disp=base1 , index=R4, scale=4

＋付加モード　BI=11, disp=base2 , index=R5, scale=1

@(R3[:B], base1[:N], R4*4[:A], R5*2[:N])[:A]

mem[R3 + base1 + R4*4 + R5*2]

R3相対付加モード

＋付加モード　BI=00, disp=base1 , index=R4, scale=4

245 246

```
    + 付加モード    EI=10, disp=base2 , index=R5, scale=2
  @(R3[:B], base1:A, R4*4:A):A
      mem[R3 + base1 + R4*4]
          R3相対付加モード
    + 付加モード    EI=00, disp=base1 , index=0, scale=*
    + 付加モード    EI=00, disp=0, index=R4, scale=4
    + 付加モード    EI=10, disp=0, index=0, scale=*
```

これは一段の付加モードで実現可能であるにもかかわら　10＊にまとめておく。ただし、括弧を省略する略記法、およ
ず、フォーマットを指定してわざわざ三段の付加モード　　び各部分を区切るコンマ’，’についてのシンタックス
にしている例である。付加モードのシンタックスを以下＊　　は、ここには含めていない。

```
オペランド = mem[mem[...] + disp + Rn *  Scale1 + Rm * Scale2...]
```

【０３１１】

```
< 一般オペランド> ::= .
      <オペランド値>[:N][.<サイズ>]
| |   <付加モードオペランド値>[.<サイズ >]

< 付加モードオペランド値> ::=
      @(<付加モード中間値>.[<disp値>[:N ]],[<インデクス値>[:N]])[
     :A]
                                          EI=10 に対応
| |   @(@(<付加モード中間値>.[<disp値>[ :N]],[<インデクス値>
                                          [:N]])[:N])[:A]
        EI=11に対応
      これは付加モード最終段を表わす。

< 付加モード中間値> ::=
      <付加モード中間値>,<disp 値>[:A]
| |   <付加モード中間値>,[<disp値>[:N]] ,< インデクス値>[:A]
                              EI=00 に対応
| |   @(<付加モード中間値>,[<disp値>[:N ]],[<インデクス値>[:N]])[:A
    ]
                              EI=01 に対応
```

これは途中にある一段の付加モードを表わす。　　【０３１２】

```
< 付加モード中間値> ::=
    [0[:B]]        絶対付加モードに対応
| |   <レジスタ番号>[:B]   レジスタ相対付加モードに対応
| |   PC[:B]              PC 相対付加モードに対応
```

これはベースモード（レジスタ相対付加モード、ＰＣ相　　対付加モード、絶対付加モードの区別）を表わす。

247 248

<disp 値> ::=
<ディスプレースメント>[:<フォーマット>]
D.dddd フィールドに対応

<フォーマット> ::= 4 | | 16 | | 32 | | 64

<インデクス値> ::=
<レジスタ番号>[.<サイズ>]['*'<スケール値>]
| | PC[.<サイズ>]['*'<スケール値>]
S,M,Rx,XX フィールドに対応

<サイズ> ::= W | | L

<スケール値> ::= 1 | | 2 | | 4 | | 8

'*'は、アスタリスク'*'を文字として使うことを示す。「繰り返し」を表わすメタ的な意味はもたない。〈インデクス〉の〈サイズ〉は、インデクスレジスタの有効なデータサイズである。本発明装置６４で'．W'を指定した場合には、レジスタの下位３２ビットが６４ビットに符号拡張される。〈インデクス〉の〈スケール〉を省略した場合には、１が仮定される。

【０３１３】Ａ２－３－６．特殊オペランド

一般のアドレッシングモード以外の方法で指定されるオペランド（特殊オペランド）に関しては、次のようなシンタックスとする。なお、各部分を区切るコンマ','についてのシンタックスは、ここには含めていない。

ｒｅｇｌｉｓｔ （ＬＤＭ，ＳＴＭ，ＥＮＴＥＲ，ＥＸＩＴＤ命令）

〈レジスタ番号〉または〈レジスタ番号〉－〈レジスタ番号〉を','で区切って並べ、'（．．．）'でくくったもの

<特殊オペランド> ::=
({<連続レジスタ番号>,} *)

<連続レジスタ番号> ::=
<レジスタ番号>
その番号のレジスタを指定
| | <レジスタ番号>-<レジスタ番号>
間の番号のレジスタをすべて指定

例：

```
ENTER.W #10,( )
LDM.W  @block,(SP)
STM.W  (R1,R3,R9-R13,FP),@-SP
```

newpc （BRA,Bcc,BSR,ACB,SCB 命令）

アドレッシング方法は、ＰＣ相対モードのみである。オペランドとしては、単にジャンプ先のラベルのみを書く。この場合は、アセンブラによって、その命令の先頭アドレスとジャンプ先のアドレスとの差がｎｅｗｐｃのビットパターンとして設定され、命令実行時にそのラベルの場所へジャンプできるようになる。

<特殊オペランド> ::=
<行き先ラベル>

【０３１４】

例：

```
BEQ    nextaddr    nextaddr にジャンプ
ACB.B  #1,R1,@limit.loopaddr
                   loopaddr にジャンプ
```

ＢＲＡ，Ｂｃｃ，ＢＳＲ，ＡＣＢ，ＳＣＢ命令は、出現頻度が高い、特殊なアドレッシング（ＰＣ相対のみ）である、慣用的に行き先のラベルをそのまま書ける方がよい、といった理由から、〈行き先ラベル〉を書くだけで自動的に〈行き先ラベル〉のアドレスと、これらの命令の置かれたアドレスとの差がディスプレースメントに設定されるようになっている。すべてのオペランド表記のうちで、レジスタ以外のシンボル名が先頭に'#'も'@'もなく現われるのは、この〈行き先ラベル〉に限られる。したがって、例えば

```
BRA    label
```

は

```
JMP    @(label-$.PC)
```

と同じ意味を表わすことになる。なお、'$'は、この記号を含む命令（この場合はＪＭＰ命令）の先頭アドレスを表わす。

249

adj （UNPKss命令）
頭に'#' を付ける。

〈特殊オペランド〉::=
| |　#〈補正値〉　　値がそのまま設定される

例：
UNPKBW　@src.@dest.#H'23302330

vector（TRAPA 命令）
頭に'#' を付ける。

〈特殊オペランド〉::=
| |　#〈ベクトル〉　　値がそのまま設定される

例：
TRAPA　#1

【０３１５】その他
・ビットフィールド命令などのリテラル指定は、短縮形のリテラル指定と同じように#〈リテラル値〉で表わす。
・CHK，INDEX，ACB，SCBビットフィールド命令などのレジスタ指定は、一般のアドレッシングにおけるレジスタ直接モードと同じように、〈レジスタ番号〉で表わす。
【０３１６】Ａ２－４．フォーマット別ニモニックと総称ニモニック
「総称ニモニック」と「フォーマット別ニモニック」は、本発明装置アセンブラの特徴の一つである。従来のプロセッサでも一部の命令について似たような考え方は見られたが（６８０２０のＭＯＶとＭＯＶＱなど）、本発明装置では両方のニモニックを完全に体系化し、オペレーションだけではなくオペランドの記述にも同じ考え方を取り入れた点に特色がある。フォーマット別ニモニックと総称ニモニックの間には、次のような関係がある。
・インプリメントやフォーマットから来るいろいろな制約をユーザに押し付けないのが総称ニモニックであり、総称ニモニックを書く限りアセンブラが最適なコードを選ぶ。同じ機能を持ち、同じようにフラッグのセットされる命令は、できる限り一つの総称ニモニックに統合する。
・フォーマット別ニモニックは、機械語のビットパターンと１対１に対応するものである。フォーマット別ニモニックが変わっても、それはオブジェクトサイズや実行サイクル数にのみ影響し、ユーザから見た命令機能はフラッグ変化まで含めて全く同じである。この点でフォーマット指定子とサイズ指定子は根本的に異なる。サイズ指定子の場合は、演算サイズが変わるとユーザから見た命令機能も変化する。条件ジャンプ命令ではＢＲＡ　ｌａｂｅｌ：３２のようなフォーマット指定子を使っているのに、加算命令の場合は、ＡＤＤ　ｓｒｃ．Ｂ，ｄｅｓｔ．Ｗのようなサイズ指定子を使っているのは、このためである。

250

・通常ユーザが使うのは、「総称」の方である。「フォーマット別」は、仕様書でのフォーマットの説明、逆アセンブラなどの特殊な用途でしか使わない。したがって、場合によっては多少冗長な感じになることもあるが、使用目的を考えると特に問題はない。ユーザが主に使用するのは、あくまで　も総称ニモニックである。また、「総称」と「フォーマット別」は両極端の表記法に過ぎず、一部のみフォーマットを指定する中間的な記法も存在する。例えば、@（ｏｆｆｓｅｔ，ＰＣ）を付加モードで書きたいが、付加モードの各段のフォーマットはどうでもよいという場合には、

@(offset.PC):A

と記述する。「フォーマット別」と言っても、どうしてもフォーマット指定の必要な部分のみ指定を行なえばよく、実際にはそれほど長い記述にはならないのが普通である。
・「フォーマット別」から「総称」への変換　は、':X'を取ることにより機械的にできるようになっている。また、逆の変換も、フォーマットが許される範囲で適当に':X'を付ければ済むようになっている。オペランドの順序は変化しない。フォーマット別ニモニックとして、記号を変えたり、順序を変えたりする方法も考えられるが、そうすると総称ニモニックとの関係がスムーズではなくなるため、そのような方法はとっていない。（いろいろな場合分けが必要になり、かえってわかりにくくなる。拡張性もよくない。）
また、前述の@（ｏｆｆｓｅｔ，ＰＣ）:Aのように、一部のフォーマットのみ指定したい場合には、「フォーマット別」か「総称」かの区別が統一的に行なえる方が望ましい。
・結局、ユーザからの要求を受けるインタフェースが総称ニモニックであり、機械語からくる制約を受けるインタフェースがフォーマット別ニモニックである。両者の調整をとるのが、':X'のフォーマット指定文字およびアセンブラである。
・フォーマット別ニモニックと総称ニモニックを併用することの欠点は、アセンブラが複雑化することである。しかし、ユーザがフォーマットのことまで心配するよりは、アセンブラで処理できることはアセンブラで処理する方がよいと考えており、そのためにアセンブラが多少複雑になるのはやむを得ない。
・機械やフラッグ変化の異なる命令は、たとえ命令のビ

251

ットパターンが似ていたとしても別の総称ニモニックになる。

【０３１７】以上のような理由から、「フォーマット別」の方は、多少記述が長くなっても、「フォーマット別ニモニックであること」や「どのフォーマットを使うかということ」がはっきりわかるようにした方がよいと考えている。フォーマットを表わす部分をすべて'：X'に統一しているのは、このためである。なお、シンタックス中で'［．．．］'　とある部分は省略可能であるが、省略するかどうかが全体で統一されている必要はない。例えば、ある'［．．．］'を省略し、別の'［．．．］'を残してもよい。

【０３１８】Ａ２－５．言語としてのアセンブラ

これまでに述べたアセンブラ表記は、機械語のビットパターンに命令としてのニモニックを与えるための表記法であり、アセンブリ言語の核となる部分である。本発明装置では、ここまでを〈〈Ｌ０〉〉仕様とする。しかし、アセンブラを一つの言語として見た場合には、ニモニック以外にも次のような項目を規定する必要がある。これらの項目については、本発明装置のアーキテクチャと矛盾を起こさない範囲で、できるだけＩＥＥＥに合わせるように標準化する。

・大文字と小文字の使用をどうするか・シンボルの文字数は何文字までにするか
・シンボルに式が書けるかどうか、シンタックスはどうなるか
・ラベルの形式はどうするか（ラベルの後に'：'をつけるかどうか）
・２進数、８進数、１０進数、１６進数の表記法はどうするか
・コメントの表記法はどうするか
・ストリングの記述形式
・特殊キャラクタ（改行文字'¥ｎ'など）の表現形式
・細かいシンタックスや使用できる文字
・アセンブラ擬似命令
・マクロ

このうち、２進数、８進数、１０進数、１６進数の表記法については、ＩＥＥＥでは以下のような形式になっている。

B'～	２進数	例:B'00010010 = H'12
Q'～	８進数	例: Q'22 = H'12
D'～	10進数	例: D'18 = H'12
H'～	16進数	

本仕様書でも、１６進数を表わすために"Ｈ"～"を、２進数を表わすために"Ｂ"～"を使用している。

【０３１９】Ａ２－５－１．大文字と小文字について

これは

252

＊ＩＥＥＥでは特に決まっていない。本発明装置では、ニモニックや予約名に対しては、大文字と小文字を同等に扱う。すなわち、この資料で大文字になっている部分に対して、小文字を使用しても構わない。ただしユーザが定義した一般の変数においては、大文字と小文字を区別する方を標準とする。

Ａ２－５－２．シンボル値

〈ディスプレースメント〉、〈リテラル値〉、〈イミディエート値〉、〈絶対アドレス〉などの項目（総称して〈シンボル値〉と呼ぶ）には、定数、ラベルを含む四則演算式が書けるものとする。式の中では、優先順位を変えるために'（．．．）'を使用することができる。ただし、未確定の値（外部名や後で定義されるラベルなど）を含む式に関しては、正しいリロケーションができるように、演算式の形を制限しても構わない。さらに、式の中では現在注目している命令のアドレスを示す値として、'＄'を使用することができる。ＰＣ相対間接モードの表記は

@(disp.PC)

であり、ｄｉｓｐの値が直接ディスプレースメントに設定される。しかし、ＰＣ相対でリロケータブルなプログラムを書く場合には、オペランドのアドレスそのものをｄｉｓｐとして設定するのではなく、オペランドのアドレスとこの命令のあるアドレスとの差をｄｉｓｐ値として設定する必要がある。この目的で'＄'を使用することができる。すなわち、（オペランド－＄）をｄｉｓｐ値として設定すればよい。

【０３２０】

'$'を使ったプログラム例

```
      <<アドレス>>
      H'00FE        ...
      H'0100        MOV.B  #1,@(loc-$:16,PC)
      H'0104        MOV.B  #2,@(8:16,PC )
      H'0108        ...
      H'010C        ...
                    ...
      H'0180  loc:  ...
```

アドレスＨ'０１００のＭＯＶ．Ｂ命令の第二オペランド@（ｌｏｃ－＄：１６，ＰＣ）では、実際のｄｉｓｐのビットパターンに設定される値がＨ'０１８０－Ｈ'０１００＝Ｈ'００８０となる。この命令により、アドレスＨ'０１８０のｌｏｃに１がセットされる。一方、Ｈ'０１０４のＭＯＶ．Ｂ命令では、アドレスＨ'０１０４＋８＝Ｈ'０１０Ｃに２がセットされる。

【０３２１】付加モードと'＄'を使ったオペランドの表記

＊

```
@(@([0:B,] label1-$[:N],PC[:N])[:A],label2-$[:N],PC[:N])[:A]
```

```
mem[mem[disp1 + PC] + disp2 + PC]
```

253

を表わす。ただし、
disp1 はlabel1の指すアドレスと現在アドレスとの差
disp2 はlabel2の指すアドレスと現在アドレスとの差

であり付加モードの拡張部は
絶対付加モード
+ 付加モード　BI=01, disp=disp1, index=PC, scale=1
+ 付加モード　BI=10, disp=disp2, index=PC, scale=1

という構成になる。このモードは、リロケータブルなテーブル（ｃａｓｅ文用のジャンプテーブルなど）をプログラム領域に置く場合に使用できる。一段目のＰＣ相対間接

mem[disp1 + PC]

は、ｃａｓｅ文用のテーブル参照をリロケータブルにするために用いられる。また、二段目のＰＣ相対間接

mem[mem[...] + disp2 + PC]

は、ジャンプ先アドレスの決定をリロケータブルに行なうために用いられる。

【０３２２】付録３．本発明装置メモリ管理方式概要

本発明装置を組み込み用などの用途に用いることを考えると、命令セットは本発明装置になっているが、メモリ管理のハードウエア（MMU）は持たないといったバージョンのチップも考えられる。したがって、本発明装置のメモリ管理機構は、必ずサポートの必要な〈〈Ｌ０〉〉仕様ではなく、標準的な仕様の掲示のみを行なう〈〈ＬＡ〉〉仕様となっている。以下では、〈〈Ｌ＊Ａ〉〉仕様としての本発明装置標準メモリ管理方式を説明する。

254

【０３２３】Ａ３－１．メモリ管理方式の選択と〈〈ＬＩＲ〉〉仕様

本発明装置では、ハードウエアによるアドレス変換とメモリ管理方式（以下MMUと呼ぶ）の標準仕様が、〈〈ＬＡ〉〉仕様として用意されている。しかし、本発明装置にＩＴＲＯＮやμＢＴＲＯＮを実装するような場合には、MMUが不要となることが多い。また、MMUを使用する用途であっても、ページテーブルなどMMU関係の実行環境の設定が終わるまでは、アドレス変換なしで命令を実行する必要がある。そこで、本発明装置では、MMU機構を使用しているかどうか、アドレス変換を行なっているかどうか、を示すフィールドをＰＳＷ内に設け、このフィールドを書き換えることにより、アドレス変換やメモリ保護の有無を指定できるようにしている。このフィールドをＡＴ（Ａｄｄｒｅｓｓ　Ｔｒａｎｓｌａｔｉｏｎ）フィールドと呼ぶ。ＡＴはＰＳＳのｂｉｔ６～ｂｉｔ７に配置されている。ＡＴをＰＳＷ内に設けたことにより、ＬＤＣＴＸ等によるコンテキストスイッチや、ＥＩＴ処理の起動、ＲＥＩＴ命令によるタクサ処理ハンドラからのリターンの際にも、アドレス変換の有無を切り換えることが可能である。ＡＴフィールドの意味を以下に示す。

【０３２４】

【表２】

AT	意　味
00	アドレス変換、メモリ保護なし
01	本発明装置標準のアドレス変換とメモリ保護<<LA>>
10	アドレス変換なし、アドレスのみを使った簡単なメモリ保護<<L1R>> 〈アドレスのMSB によるメモリ領域の区別、 2リング〉
11	reserved

【０３２５】このうち、〈〈ＬＡ〉〉仕様の標準メモリ管理を実装した本発明装置ではＡＴ＝００，０１が利用可能、〈〈Ｌ１Ｒ〉〉本発明装置ではＡＴ＝００，１０が利用可能となる。〈〈Ｌ１Ｒ〉〉仕様のＡＴ＝１０の場合、MMUがないのでページ毎のメモリ保護は行なえないが、〈〈ＬＡ〉〉の４リングを縮退してリング０、リング３のみを有効とし、アドレスによって簡単なメモリ保護を行なう。アドレスのＭＳＢ＝１の領域（〈〈ＬＡ〉〉でいうＳＲ）はリング０からアクセス可能、リング３からアクセス禁止の領域であり、通常はここにＯＳを置く。一方、アドレスのＭＳＢ＝０の領域（〈〈ＬＡ〉〉でいうＵＲ）はリング０からもリング３からもアクセス可能な領域であり、通常はここにユーザプログラムを置く。MMUがないのでユーザプログラム間のメモリ保護はできないが、ユーザプログラムからＯＳを保護することは可能である。ＡＴ＝００（アドレス変換なし）の場合は、メモリアクセスに対するリング保護のチェックは行なわれない。したがって、ページ不在例外（ＰＯＥ）、アドレス変換例外（ＡＴＲＥ）は発生しない。ただし、ＡＴ＝００の場合も特権命令のチェックは行なわれる。ＡＴ＝００の時の動作については、〈〈Ｌ１〉〉と〈〈Ｌ１Ｒ〉〉で全く同じになることが望ましい。しかし、ＬＤＡＴＥなどの命令では、〈〈Ｌ１〉〉ならばMMU環境の設定という意味で実用的な命令であるのに対して、〈〈Ｌ１Ｒ〉〉では全く意味を持たない。また、ＰＴＬＢなどの命令も〈〈Ｌ１〉〉のＡＴ＝

255

００ならば一応意味を持っているが、〈〈Ｌ１Ｒ〉〉ではＴＬＢそのものがないので全く意味を持たない。したがって、〈〈Ｌ１Ｒ〉〉仕様ではこういったMMU関係の命令を実装しないことにする。〈〈Ｌ１Ｒ〉〉でこういった命令を実行しようとした場合は、ＡＴの値にかかわらず予約命令例外（ＲＩＥ）となる。

【０３２６】Ａ３－２．本発明装置のメモリ管理方式

本発明装置は〈〈Ｌ１Ｒ〉〉仕様のチップである。本発明装置のＡＴフィールドの意味を図２９７に示す。

【０３２７】Ａ３－３．本発明装置のＩ／Ｏ空間アクセスに関して

ＩＯＭＡＳＫ，ＩＯＡＤＤＲで示されるＩ／Ｏ空間に対する命令フェッチ及びメモリ間接アドレッシング・モードによるオペランドフェッチはアドレス変換例外となる。Ｉ／Ｏ空間に対するアクセスで、メモリ間接アドレッシングの場合はアクセス動作は一切行なわれない。しかし、命令フェッチの場合にはアクセス動作が行なわれる。そのため、外部のＩ／Ｏデバイスはバスアクセスタイプ（ＢＡＴ）信号をみて命令フェッチであれば応答しないようにする必要がある。Ｉ／Ｏ空間は通常リング０の領域におかれるためリング３からのアクセスはリング保護違反が検出されることになると考える。リング保護違反の場合は高速に検出可能なためメモリアクセスは行なわれない。また、Ｉ／Ｏ空間とＩ／Ｏでない空間をまたぐようなアクセスが行われた場合、アドレス変換例外を起こすが、その場合の再実行動作を保証できない。

【０３２８】Ａ３－４．メモリ管理の目的と概要

本発明装置では、以下のような目的を達成するために、ハードウエアによるメモリ管理機構（MMU）を導入している。

・命令再実行と仮想記憶方式のサポートにより、実装されている物理メモリの量を越える大きさの論理空間を提供する。

・多重論理空間の機能の導入により、コンテキスト（タスクやプロセス）間の独立性を維持し、プログラムを作りやすくする。

・リング保護の機能の導入により、ＯＳや共有データとユーザプログラムやユーザデータとの間でメモリ保護を行なう。

本発明装置では、以上のような機能を提供するために、毎回のメモリアクセスでページング方式によるアドレス変換を行なう。アドレス変換前のアドレス（論理アドレス）の作る空間を、Ｌｏｇｉｃａｌ　Ｓｐａｃｅの意味でＬＳと予備、アドレス変換後のアドレス（物理アドレス）の作る空間を、ＰｈｙｓｉｃａｌＳｐａｃｅの意味でＰＳと呼ぶ。ページングの場合には、メモリアクセスを高速化するために、ＴＬＢ（Ｔｒａｎｓｌａｔｉｏｎ　Ｌｏｏｋａｓｉｄｅ　Ｂｕｆｆｅｒ）と呼ばれるアドレス変換対の記憶バッファを導入することが一般的である。しかし、ＴＬＢはメモリアクセスを高速化するため

256

のインプリメント上の手段と考えられるため、本発明装置のMMU仕様としてはＴＬＢに関する規定は行なわない。本資料でも、ＴＬＢに関する説明は行なっていない。また、コンテキストの切り換えによるＴＬＢのパージを減らすため、〈〈Ｌ２〉〉として論理空間識別子（ＬＳＩＤ）の機能を導入することができる。ＬＳＩＤは、コンテキスト毎にユニークに与えられた番号であり、ＬＳＩＤまで含めてＴＬＢの論理アドルス比較を行なうようにすれば、コンテキストを切り換える際にもすべてのＴＬＢをパージする必要がなくなる。しかし、ＬＳＩＤの機能についてもインプリメント依存性が強いため、本発明装置のMMU仕様としては、ＬＳＩＤの詳細機能やビット数の規定を行なっていない。本発明装置の仕様では、ＬＳＩＤを示す制御レジスタのアドレス割り当てと、ＬＳＩＤを実装した場合のＴＬＢやキャッシュの整合性確保に関する説明のみが行なわれている。

【０３２９】Ａ３－４－１．ページング

本発明装置のアドレス変換は、ページングを基本としている。ページサイズは４ＫＢとして本発明装置全体で統一されている。これによって、ＴＬＢの構造などをある程度限定することができ、メモリ管理機構を内蔵できるチャンスが大きくなる。また、基本的なパラメータを固定化すれば、それにチューニングすることで性能向上が期待できる。アドレス変換に際し、本発明装置３２の論理アドレスは図２９８のように分割され、これによって２段階のページングを行なう。３２ビットの論理アドレスは、４ＧＢの論理空間を作る。Ｒビット（論理アドレスのＭＳＢ）により、４ＧＢの論理空間は２ＧＢのＵｎｓｈａｒｅｄ　Ｒｅｇｉｏｎ（ＵＲ）とＳｈａｒｅｄ　Ｒｅｇｉｏｎ（ＳＲ）に分かれる。おのおののＲｅｇｉｏｎは、ＳＸフィールドにより４ＭＢずつのＳｅｃｔｉｏｎに分かれる。さらに、おのおののＳｅｃｔｉｏｎは、ＰＸフィールドによりＲＫＢずつのＰａｇｅに分かれる。したがって、２段階のページングのうち上位段のページテーブルは、ＳＸをインデクスとして、下位段のページテーブルのベースアドレスを引き出すものとなる。これをＳｅｃｔｉｏｎ　Ｔａｂｌｅ（ＳＴ）と呼び、その一つのテーブルエントリをそのエントリをＳＴＥと呼ぶ。また、２段階のページングのうち下位段のページテーブルは、ＰＸをインデクスとして、物理ページのベースアドレスを引き出すものとなる。これをＰａｇｅ　Ｔａｂｌｅ（ＰＴ）と呼び、その一つのテーブルエントリをそのエントリをＰＴＥと呼ぶ。一つのＳＴＥの変更により一つのＳｅｃｔｉｏｎが、一つのＰＴＥの変更により一つのＰａｇｅが影響を受けることになる。ＰＴＥ，ＳＴＥの総称名としてＡＴＥ（Ａｄｄｒｅｓｓ　Ｔｒａｎｓｌａｔｏｎ　ｔａｂｌｅ　Ｅｎｔｒｙ）という名称を用いる。ＳＴのベースアドレスは、ＵＲの場合ＵＡＴＢ、ＳＲの場合ＳＡＴＢという制御レジスタによって示される。ＵＡＴＢまたはＳＡＴＢの変更により、

257

一つのＲｅｇｉｏｎ（ＵＲ又はＳＲ）が影響を受ける。以上述べた関係をまとめたものを図２９９に示す。

【０３３０】Ａ３－４－２．多重論理空間

本発明装置では、論理アドレスのＭＳＢによって、共通空間（ＳｈａｒｅｄＲｅｇｉｏｎ）と個別空間（Ｕｎｓｈａｒｅｄ　Ｒｅｇｉｏｎ）が区別されている。それぞれのＲｅｇｉｏｎに対して、アドレス変換のテーブルベースレジスタＵＡＴＢ，ＳＡＴＢが存在するが、このうちＵＡＴＢのみはコンテキスト毎に切り換わるようになっており、これによって多重論理空間を実現する。すなわち、ＵＲ（論理アドレスＨ’００００００００～Ｈ’ｆｆｆｆｆｆｆｆ）ではコンテキスト毎に別々の物理空間（物理メモリ）が割り当てられるのに対して、ＳＲ（論理アドレスＨ’８０００００００～Ｈ’ｆｆｆｆｆｆｆｆ）ではコンテキスト間で共通の物理空間（物理メモリ）が割り当てられる。Ｓｈａｒｅｄ　Ｒｅｇｉｏｎは主に割り込み処理ハンドラやＯＳが使用し、Ｕｎｓｈａｒｅｄ　Ｒｅｇｉｏｎは主にユーザプログラムが使用するものであるが、ユーザデータでもタスクやプロセス間で共用するものはＳＲを使用する場合があるし、ＯＳの管理するデータでもタスクやプロセス毎に持つ必要のあるものはＵＲを使用する場合がある。多重論理空間の機能を利用すれば、同一の論理アドレスから複数のプログラムを同時に実行することができるので、実行時にオブジェクトコードのリロケーションをする必要がなくなる。また、他のタスクやプロセスのＵＲを直接参照することはできないので、プログラム間のメモリ保護にも役立つ。なお、ＵＲ，ＳＲの区別と後述のメモリのリング保護の機能とは、直接の関連はない。すなわち、ｒｉｎｇ３からＳＲを参照できないとか、ＰＣがＵＲにある間はＳＲを参照できないとかといった制限は、ＵＲ，ＳＲ自体の機能には含まれていない。このようなメモリのアクセス制限を行ないたい場合には、メモリのリング保護の機能を利用し、ＳＴＥ，ＰＴＥに適当な保護コードを設定しておかなければならない。

【０３３１】〔多重論理空間の構成〕図３００において

・Ｕｎｓｈａｒｅｄ　Ｒｅｇｉｏｎ／Ｓｈａｒｅｄ　Ｒｅｇｉｏｎの切り替えは、論理アドレスのＭＳＢによって行なわれる。

・Ｕｎｓｈａｒｅｄ　Ｒｅｇｉｏｎでは、ＵＡＴＢレジスタによってアドレス変換される。

・Ｓｈａｒｅｄ　Ｒｅｇｉｏｎでは、ＳＡＴＢレジスタによってアドレス変換される。

・ＵＡＴＢレジスタのみコンテキスト毎に切り換わるため、ＵｎｓｈａｒｅｄＲｅｇｉｏｎでは、それぞれのコンテキストが別々の論理空間を持つことができる。アドレス変換が２段のページングとなっているため、異なるコンテキストの２つのＳＴＥが同一のＰａｇｅ　Ｔａｂｌｅを指すことにより、ＵＲの中での共用セクション、共有ページを設けることもできる。なお、ＵＲとＳＲの

258

境界にまたがるようなプログラムやデータについては、以下のように考える。

・６４ビット拡張時には、現在連続しているＵＲとＳＲの境界が不連続になるため、Ｈ’７ｆｆｆｆｆｆｆｆから先に伸びているプログラムは、６４ビット拡張時に使用できなくなる。これは、本発明装置３２で、Ｈ’７ｆｆｆｆｆｆｆｆの次のアドレスをＨ’００００００００と考えてもＨ’８０００００００と考えても同じである。したがって、ＵＲとＳＲの両方にまたがるようなメモリアクセスや、ＵＲ～ＳＲで連続するような命令のフェッチを行なうべきではない。

・しかしながら、ＵＲとＳＲにまたがっているかどうかのチエックを毎回行なうのは、インプリメント上の負担が大きいため、ＵＲとＳＲの両方にまたがるアクセスがあったとしてもＥＩＴとはしない。この場合、Ｒｅｇｉｏｎとは関係なく、アドレスがリニアアドレスであるという考え方を生かし、Ｈ’７ｆｆｆｆｆｆｆｆの次はＨ’８０００００００、Ｈ’ｆｆｆｆｆｆｆｆの次はＨ’００００００００のアドレスをアクセスするものとする。ただし、前項でも述べたように、この仕様を利用するようなプログラムを書くべきではない。ＰＳＷ，ＵＡＴＢ，ＳＡＴＢに対するＬＤＣ命令や、ＬＤＡＴＥ，ＬＤＣＴＸ命令による論理空間の切り換えでは、現在実行中のプログラム領域のアドレス変換に対しても影響を与えることができる。したがって、使い方によっては次の命令から全く別の場所に飛んでしまうというケースも生じる。これはプログラムの責任で処理しなければならない。具体的には、ＡＴビットの変更を行なう場合はＶ＝Ｒ領域を利用したり、ＬＤＣＴＸを実行する場合はＳＲ（Ｓｈａｒｅｄ　Ｒｅｇｉｏｎ）を利用したりすることになる。

【０３３２】Ａ３－４－３．リング保護

本発明装置のメモリ保護方式は、４レベルのリング保護である。保護情報は、論理アドレスやＵＲ，ＳＲの区別には関係なくページ毎に指定することができる。リング保護では、ＰＳＷ中に示される現在リング番号（ＲＮＧ）と、アクセスすべき論理アドレスのＳＴＥ，ＰＴＥに含まれている保護コードとの関係によって、アクセスが可能かどうかが決まる。ＲＮＧが小さい値であるほど、すなわち内側のリングであるほど強いアクセス権を持っており、ＲＮＧ＝３が最もアクセス権が弱い。ｒｎｇ１＜ｒｎｇ２とした場合、ＲＮＧ＝ｒｎｇ２でアクセスできるページは必ずＲＮＧ＝ｒｎｇ１でもアクセスできることになる。

【０３３３】Ａ３－５．ＭＭＵ関係制御レジスタ

ＭＭＵに関係する制御レジスタについて説明する。

・ＰＳＷ

別項参照。ＭＭＵに関連するのは、ＲＮＧ，ＡＴのフィールドである。

・ＬＳＩＤ

259

別項参照。このレジスタの有無と有効なビット数は、インプリメントに依存する。

・UATB（Unshared region Address Translation table Base）

260

図３０１に示す。

・SATB（Shared region Address Translation table Base）

図３０２に示す。

STB:　Section Table Base

Section Table の物理ベースアドレス

=:　'0' にreserved

ユーザ向けのマニュアルでは、'0' を書き込みように指導しておく。ただし、'1' を書き込んでも無視される。

読みだし時は値不定である。

D:　Direction

Section Table のサイズが小さい場合のSection Table の方向

D=0　Section Table の下位アドレス側が有効

D=1　Section Table の上位アドレス側が有効

D とLLのフィールドは、小規模な用途で、有効な論理アドレスの範囲を制限し、同時にSection Table のサイズを縮小するために使用するものである。

LL:　Length

Section Table のサイズ

LL=00	1/2 サイズ	512 エントリ	2KBテーブル	2GB Region
LL=01	1/4 サイズ	256 エントリ	1KBテーブル	1GB Region
LL=10	1/16サイズ	64 エントリ	256Bテーブル	256MB Region
LL=11	1/64サイズ	16 エントリ	64Bテーブル	64MB Region

PI:　Page In

PI=0　Section Table が物理アドレスに存在しない。

この場合には、STB はハードウエア的な意味を持たないので、OSで自由に使用することができる。ただし、PI=0でもD, LL のフィールドは有効である。D, LL のフィールドは、アドレス変換例外(ATRE)の未使用領域参照エラーか、ページ不在例外(PGE) かを区別するために使用され、例外の検出は前者が優先される。

つまり、PI=0の時にこのレジスタを使ってアドレス変換を行おうとした場合、D,LLのフィールドにより未使用領域参照エラーが検出されればアドレス変換例外(ATRE)が発生し、そうでなければページ不在例外(PGE) が発生する。

PI=1　Section Table が物理アドレス上に存在する。

PI=1であれば、このレジスタを使ってアドレス変換を行なうことが可能である。ただし、D, LL のフィールドにより未使用領域参照エラーが検出された場合には、アドレス変換例外(ATRE)が発生する。

261

【０３３４】ＳＡＴＢ，ＵＡＴＢ中のＰＩビットは、正確にはＰａｇｅ　Ｉｎではなく Ｓｅｃｔｌｏｎ　Ｔａｂｌｅ　Ｉｎの意味を表わす。また、後述するＳＴＥ中のＰＩビットは、Ｐａｇｅ　ＩｎではなくＰａｇｅ　ＴａｂｌｅＩｎの意味を表わす。しかし、あえて区別する必要もないと思われるので、いずれも同じ’ＰＩ’という名称を使用する。広義の「ページ」は、「ページテーブル」や「セクションテーブル」、つまりディスクへの追い出しの単位となるものをすべて含むことになり、ページテーブルやセクションテーブルの不在にも「ページ不在例外」という例外の名称をそのまま適用する。Ｓｅｃｔｉｏｎ　Ｔａｂｌｅ，Ｐａｇｅ　Ｔａｂｌｅ自体をページアウトすることも可能であるが、ページインされているテーブルについては、テーブルのベースアドレス（ＳＴＢ，ＰＴＢ）は物理アドレスを表わすものとする。すなわち、ページテーブルは論理空間ではなく物理＊

262

＊空間に置かれていると考えることができる。また、Ｓｅｃｔｉｏｎ　ＴａｂｌｅやＰａｇｅ　Ｔａｂｌｅが４Ｋのフルサイズでない場合にＳＴ，ＰＴのページアウトを許すと、６４Ｂ，２５６Ｂ，１ＫＢ，２ＫＢ単位でページ不在となるケースも生じる。つまり、ＵＡＴＢ，ＳＡＴＢ，ＳＴＥでのＰＩビットは、必ずしも４Ｋ単位でのページ不在を指すとは限らないので、注意する必要がある。セクションテーブル、ページテーブルの大きさが可変になっているが、テーブルが最小（１／６４）の大きさでない時には、ＳＴＢとＳＸの有効なビットに重なりが生じる。例えば、ＵＡＴＢ，ＳＡＴＢでＤ＝０，ＬＬ＝００とした場合、以下のような計算によりＳＴＥの実効アドレスを算出する必要がある。

【０３３５】

【数１７】

X は有効なビット(0/1)

- は無効なビット(0)

'>>'はシフトを表わす

```
  STB          XXXX XXXX XXXX XXXX XXXX XXXX XX-- ----
+)SX>>20  B'   ---- ---- ---- ---- ---- ---- XXXX XX --
```

【０３３６】ここで、’Ｘ’の重なっているビット、すなわちＳＴＢの２＾６～２＾１１のビットの扱いが問題となる。以下のような案が考えられる。

①ＳＴＢの２＾６～２＾１１のビットは’０’でなければならないものとする。インプリメント上は、ＳＴＢの２＾７～２＾３１とＳＸを２０ビット右にシフトしたものとを連結することによって、ＳＴＥのアドレスを算出することができる。（アラインメント強制）

②ＳＴＢの２＾６～２＾１１のビットが０でなくてもよいが、２＾１１までの範囲でＳＴＥの実効アドレスがラップアラウンドする。インプリメント上は、重なりのある５ビットのみの加算を行ない、加算により生じた桁上がりは無視することになる。（ラップアラウンド）

③ＳＴＢの２＾６～２＾１１のビットが０でなくてもよく、ＳｅｃｔｉｏｎＴａｂｌｅに対して２＾６より上位のアラインメントは全く強制しない。インプリメント上は、まず重なりのある５ビットの加算を行ない、桁上がりを生じた場合には最上位桁まで桁上がりを伝搬させる必要がある。つまり、２＾６より上位のビットについてすべて加算を行なう必要がある。（アラインメント自由）

【０３３７】現在③が本発明装置の仕様となっている。ＬＬ＝０１，１０の場合、またＰＴＢとＰＸのアラインメントの場合も同じように③の仕様とする。ＤとＬＬの機能は、小規模な用途でＳｅｃｔｉｏｎ　Ｔａｂｌｅの領域を節約するために設けられているものである。ＤとＬＬの指定が変わると、Ｓｅｃｔｉｏｎ　Ｔａｂｌｅが小さくなるため、ＳＸの取り得る値に制限ができる。Ｄ，ＬＬのそれぞれの値に対して、許されているＳＸの値は図３０３のようになる。Ｄ＝１，ＬＬ＝００のｒｅｓｅｒｖｅｄの部分は、ソフトウェアで使用してはいけない。マニュアルにもｒｅｓｅｒｖｅｄであることを明記する必要がある。ただし、この値を実際にＵＡＴＢやＳＴＡＢに設定した場合は、Ｄ＝０，ＬＬ＝００と同じ動作をする。つまり、ＬＬ＝００の場合にはＤは無視される。論理アドレスとじて、上の表に当てはまらない値を指定した場合には、アドレス変更例外（ＡＴＲＥ）の未使用領域参照エラーが発生する。ページ不在例外（ＰＯＥ）とアドレス変更例外（ＡＴＲＥ）が同時に発生した場合には、アドレス変更例外（ＡＴＲＥ）が優先される。例えば、ＵＡＴＢのＤ＝０，ＬＬ＝１０，ＰＩ＝０で論理アドレスＨ’４０００００００をアクセスした場合には、ＰＯＥではなくＡＴＲＥとなる。ＤとＬＬの指定によるテーブル領域の節約のようすを図示すると、図３０４のようになる。この時、使用する論理アドレスの範囲が狭いため、はじめの数個のＳＴＥのみが利用できれば済むという①の例であれば、図３０５のようにすることによって、ｓｅｃｔｉｏｎ　ｔａｂｌｅの大きさを節約することができる。又、スタック領域やＯＳ用の領域などで、終わりの数個のＳＴＥのみが利用できれば済むという②の例であれば、図３０６のようにすることによって、ｓｅｃｔｉｏｎ　ｔａｂｌｅの大きさを節約することができる。

※Ｄ＝１の場合、ＳＴＢはｓｅｃｔｉｏｎ　ｔａｂｌｅの有効部分の先頭アドレス（ＳＸとしては途中の値に対応する）を指すのではなく、ＳＸ＝０に対応するＳＴＥ

263

（実際には存在しない）の置かれるべきアドレスを指す。ＳＴＥ中のＰＴＢの場合も同様である。

【０３３８】Ａ３－６．ＳＴＥとＰＴＥ

ＳＴＥとＰＴＥは、メモリ上に置かれるアドレスの変換

264

用のディスクリプタである。ＳＴＥとＰＴＥのフォマットについて説明する。

・ＳＴＥ（Ｓｅｃｔｉｏｎ　Ｔａｂｌｅ　Ｅｎｔｒｙ）

図３０７に示す。

PTB: Page Table Base

Page Table の物理ベースアドレス

W: Write可能

W=1 書き込みのアクセス権はPTE の保護コードに従う

W=0 PTE の保護コードにかかわらず、全リングから書き込み禁止

違反した場合はアドレス変換例外(ATRE)のリング保護違反エラーが発生

E: Execute可能

E=1 実行のアクセス権は PTEの保護コードに関わらず、全リングから実禁止

違反した場合はアドレス変換例外(ATRE)のリング保護違反エラーが発生

D: Direction

Page Table のサイズの小さい場合のPage Tableの方向

D=0 Page Tableの下位アドレス側が有効

D=1 Page Tableの上位アドレス側が有効

LL: Length

Page Table のサイズ

LL=00	フルサイズ	1024エントリ	4KBテーブル	4MB Section
LL=01	1/4 サイズ	256エントリ	1KBテーブル	1MB Region
LL=10	1/16サイズ	64エントリ	256Bテーブル	256KB Region
LL=11	1/64サイズ	16エントリ	64Bテーブル	64KB Region

PI: Page In

PI=0 Page Tableが物理アドレス上に存在しない。

この場合には、PTB はハードウエア的な意味を持たないので、OSに自由に使用することができる。ただし、PI=0でもD,LLのフィールドは有効である。D.LLのフィールドは、アドレス変換例外(ATRE)の未使用領域参照エラーやリング保護違反エラーか、ページ不在例外(PDE) かを区別するために利用され、例外の検出は前者が優先される。

つまり、PI=0の時にこのSTE を使ってアドレス変換を行なおうとした場合、D.LLのフィールドにより未使用領域参照エラーやリング保護違反エラーが検出されればアドレス変換例外(ATRE)が発生し、そうでなければページ不在例外(PDE) が発生する。

【０３３９】

265　　266

PI=1　Page Tableが物理アドレス上に存在する。
PI=1であれば、このSTE を使ってアドレス変換を行なうことが可能である。ただし、D,LLのフィールドにより未使用領域参照エラーやリング保護違反エラーが検出された場合には、アドレス変換例外(ATRE)が発生する。

ＳＴＢとＳＸの時と同様に、ＰＴＢとＰＸの場合にも、ＰＴＢのアラインメントは自由とする。すなわち、ＬＬ≠１１の時には、ＰＴＢとＰＸの有効なビットに重なりの生じる場合があるが、重なった部分のアドレスについては最上位桁まで加算を行なう。ＤとＬＬの機能は、小規模な用途でＰａｇｅ　Ｔａｂｌｅの領域を節約するために設けられているものである。ＤとＬＬの指定が変わると、Ｐａｇｅ　Ｔａｂｌｅが小さくなるため、ＰＸの取り得る値に制限ができる。Ｄ，ＬＬのそれぞれの値に対して、許されているＰＸの値は図３０８のようになる。論理アドレスとして、上に当てはまらない値を指定した場合には、アドレス変換例外（ＡＴＲＥ）の未使用領域参照エラーが発生する。Ｄ＝１，ＬＬ＝００の部分に対しては、Ｗ，Ｅが特別な意味を持つ。Ｄ＝１，ＬＬ＝００のＳＴＥを使ってメモリアクセスを行なおうとした場合には、次のような動作をする。この時、ＳＸの値は関係しない。また、ＰＴＢのフィールドとＥビットは、ハードウェアで使用しないので、ＯＳから自由に利用することができる。

【０３４０】図３０９において
’＊’はソフトウエアで自由に使用してよいビットである。ハードウエア的にはこのビットが無視される。このビットは、ｒｅｓｅｒｖｅｄを示す’＝’とは異なり、将来の使用拡張でも使用しないことがはっきりしているビットである。’＝’と’＊’は、現在の使用におけるハードウエア的な動作は同じであるが、将来の拡張のために仕様書上きの扱いが異なっている。なお、未使用領域参照エラーと予約ＡＴＥエラーの区別は、アドレス変換例外（ＡＴＲＥ）が起動された際にスタックに積まれるエラーコードによって行われる。これらのエラーによるアドレス変換例外の起動は、ＳＴＥ，ＰＴＥの値を設定した時に検出されるのではなく、値を使用する時（アドレス変換の時、つまりメモリアクセスを行った時）に検出される。Ｄ＝１，ＬＬ＝００，Ｗ＝０の機能は、一つのｓｅｃｔｉｏｎ全体を未使用領域としたい場合に利用する機能である。この場合、このＳＴＥに対するＰａｇｅＴａｂｌｅは無い。ＬＬ≠００の機能を利用して小さいサイズのＳｅｃｔｉｏｎ　Ｔａｂｌｅ，Ｐａｇｅ　Ｔａｂｌｅを使用する場合でも、論理アドレス中のＳＸ，ＰＸの位置は変わらない。したがって、複数のＳＴＥから小さいサイズのＰａｇｅ　Ｔａｂｌｅを使用する場合には、有効な論理アドレスが飛び飛びの値をとることになる。この様子を図３１０に示す。

【０３４１】このように、小さいサイズのＰＡｇｅ　Ｔａｂｌｅを使った場合には、有効な論理アドレスが連続領域とはならないことがある。しかし、ＳＴＥを１エントリしか使用しないような小規模な応用のため、あるいは、長さが半端になった論理空間の最後の部分のテーブルについて、「テーブル領域の節約」を行なうためにＬＬ≠００の機能が用意されているのだと考えると、ＬＬ≠００の時にアドレスが連続しなくても、特に問題はない。ＳＴＥでは、Ｄ，ＬＬによるアドレス変換例外（ＡＴＲＥ）の未使用領域参照エラー、Ｗ＝０，Ｅ＝０によるアドレス変換例外（ＡＴＲＥ）のリング保護違反エラー、ＰＩ＝０によるページ不在例外（ＰＯＥ）が同時に発生する可能性があるが、例外の検出順序はこの順とする。つまり、まず未使用領域かどうかのチェックを行ない、次にアクセス権のチェックを行ない、最後にページ不在のチェックを行なう。これは、ＰＴＥの場合も同様である。したがって、ページ不在の時でも、リング保護関係の情報は有効である。ただし、この例外検出順序は、一つの段（ＳＴＥまたはＰＴＥ）の中での例外検出順序であり、これよりもさらにテーブルを引く順序が優先する。つまり、，さたで発生する例外よりもＳＴＥで発生する例外の方が優先される。

【０３４２】・ＰＴＥ（Ｐａｇｅ　Ｔａｂｌｅ　Ｅｎｔｒｙ）　図３１１に示す。

267 268

PFN: Page Frame Number
対応するページの先頭物理アドレス
4KB が単位になる。

**: OSが自由に利用できるビット
ハードウエア的には意味を持たない

R: Refereoced
このページが参照されたとき、1 にセットされる。

M: Modified
このページの一部が変更されたとき、1 にセットされる。

RL: Read Level
RL=00 リング0 のみから読みだし可能
RL=01 リング0 ～1 から読みだし可能
RL=10 リング0 ～2 から読みだし可能

RL=11 リング0 ～3 から読みだし可能
違反した場合はアドレス変換例外(ATRB)のリング保護違反エラーを発生

T: Type
T=00 書き込み禁止、実行禁止
T=01 書き込み禁止、実行可能
T=10 書き込み可能、実行禁止
T=11 書き込み可能、実行可能
「書き込み可能」の場合は、min(RL, AL)のリングから書き込み可能、それより外側からは書き込み禁止となる。
「実行可能」の場合は、max(RL, AL)のリングから実行可能、それより外側からは実行禁止となる。
違反した場合はアドレス変換例外(ATRB)のリング保護違反エラーを発生

【０３４３】

269 270

AL: Access Level for indicated type

(T ≠00の場合)

AL=00 リング0 のみからアクセス（書き込み、実行）可能

AL=01 リング0 ～1 からアクセス（書き込み、実行）可能

AL=10 リング0 ～2 からアクセス（書き込み、実行）可能

AL=11 リング0 ～3 からアクセス（書き込み、実行）可能

(T=00の場合)

AL=00 リング0 ～RLから読みだし可能

書き込み、実行は禁止(T.RL の本来の意味で使用)

AL=01 アドレス変換例外(ATRE)の未使用領域参照エラーを発生

AL=10 (reserved)

AL=11 (reserved)

T=00の場合は、ALによるアクセス可能リングの指定が必要ないので、AL を別の意味に使っている。このうち、AL=01 の指定による未使用領域参照エラー発生の機能は、一つのPage全体を未使用領域としたい場合に利用する機能である。この場合、このPTE に対する物理ページは存在しな

271　　272

い。

NC:　Non Cachable

NC=1　キャッシュへの取り込みの禁止を指定

I/O レジスタやVRAMの領域などで、キャッシュへの取り込みやメモリアクセスの順序の変更が許されないページの場合に、このビットをセットする。

NC=0　通常のページであるたとを指定

PI:　Page In

PI=0　Pageが物理アドレス上に存在しない。

この場合には、PFN はハードウエア的な意味を持たないので、OSに自由に使用することができる。ただし、PI=0でもRL,T,AL などのフィールドは有効である。これらのフィールドは、アドレス変換例外(ATRE)の未使用領域参照エラーやリング保護違反エラーか、ページ不在例外(POE) かを区別するために利用され、例外の検出は前者が優先される。

つまり、PI=0の時にこのPTE を使ってアドレス変換を行おうとした場合、RL,T,AL などのフィールドにより未使用領域参照エラーやリング保護違反エラーが検出されればアドレス変換例外(ATRE)が発生し、そうでなければページ不在例外(POE) が発生する。

PI=1　Page Tが物理アドレス上に存在する。

PI=1であれば、このPTE を使ってアドレス変換を行なうことが可能である。

ただし、RL,T,AL などのフィールドにより未使用領域参照エラーやリング保護違反エラーが検出された場合には、アドレス変換例外(ATRE)が発生する。

【０３４４】ＰＦＮと論理アドレスのｏｆｆｓｅｔの間では、有効なビットが重なり合うことはないため、アラインメントの問題は発生しない。ＰＴＥのＲＬ，Ｔ，ＡＬの値と、実際にそれをアクセス可能なリングとの関係は、具体的には図３１２のようになる。

図において

・Ｒ０はｒｉｎｇ０からのアクセス権、Ｒ１はｒｉｎｇ１からのアクセス権を示す。ｒｉｎｇ０～ｒｉｎｇ３の区別はＰＳＷ中のＲＮＧフィールドで示される。

・Ｒは読みだし可能、ｗは書き込み可能、Ｅは実行可能を示す。Ｔ＝００，ＡＬ≠００の場合、ＡＬが特別な意味を持ち、図３１３のような動作をする。この時、ｏｆｆｓｅｔの値は関係しない。このうち、Ｔ＝００，ＡＬ＝０１の場合には、ハードウエアで使用しないＰＴＢのフィールドとＲＬフィールドを、ＯＳから自由に利用することができる。Ｔ＝０１を指定すると、読みだしはできないが実行は可能であるというページを作ることができる。これは、プログラムのコピーを禁止し、プログラムの実行に対する課金メカニズムを導入することを意図したものである。一方、Ｔ＝００，Ｔ＝１０を指定すると、読みだしや書き込みは可能であるが、実行は禁止であるというページを作ることができる。この機能を利用すれば、プログラムカウンタがデータ領域に飛び込んで来た場合に、それをチェックしてプログラムの暴走を阻止することができる。実行を禁止する機能は、メモリのデータ保護のための機能というよりも、デバッグのための機能と考えることができる。書き込みが可能である場合には、からなず読みだしも可能になっている。

【０３４５】Ａ３－７．６４ビットへの拡張性

ＳＲ／ＵＲの切り換えビットを論理アドレスのＭＳＢに固定すると、６４ｂｉｔへの拡張時に問題が生じる可能性がある。本発明装置では、論理アドレスを符号付きの数と考えることによって、この問題に対処する。ＳＲとＵＲの双方を３２ビットから６４ビットに拡大するためには、アドレス空間が二方向に伸びればよいわけである。そこで、アドレスを符号付きの数と考え、ＵＲ領域が正方向にＳＲ領域が負方向に伸びると考えれば、この問題は解決される。具体的には、３２→６４の拡張に対

273

して、論理アドレスは符号拡張するようにしておく。メモリマップは図３１４のようになる。あるいは、図の書き換え方をかえて、図３１５のようになる。アドレスを符号付きと考えることにより、ＳＲ，ＵＲ双方の領域で拡張に対する連続性が保たれる。その代わり、Ｈ’８０００００００におけるＯＳ領域とユーザ領域の接点が切られることになるが、これは問題ないと考えられる。なお、本発明装置の１６ビット絶対アドレッシングモード（@ａｄｓ：１６）で、論理アドレスを符号拡張するようになっているのも、アドレスを符号付きとする考え方を適用したものである。

【０３４６】Ａ３－８．ＭＭＵ機能のバリエーションとＬＳＩＤの機能

ＬＳＩＤの機能は〈〈Ｌ２〉〉であるため、最初のチップでは実装されず、将来のチップで導入される可能性が高い。したがって、最初に出たチップのためのＬＳＩＤの機能を利用しないプログラムも、将来のＬＳＩＤの機能の実装を見越したものにしておくのが望ましい。かといって、最初のチップでも不必要なオーバーヘッドは避けなければならない。そこで、こういったＬＳＩＤ（論理空間識別子）機能の有無やＭＭＵ機能の各種のバリエーションと、プログラム（ＯＳ）の互換性との関係について、検討を行なう必要がある。一般的な話として、ＭＭＵ関係の仕様のバリエーションに対する対応には次の２つの方針がある。

①互換性を維持するため、実装していない機能であっても、縮退した機能でそのまま実行する。この場合、性能は落ちるだけで、オブジェクトレベルの互換性は達成される。

②実装していない機能については、ＥＩＴで検出する。ＥＩＴ起動の目的は、まちがいの検出とエミュレーションであるが、エミュレーションでは極端な性能の低下を招く場合があるので、実際問題として、オブジェクトレベルで変更の必要な場合が多い。

したがって、オブジェクトレベルの互換性は　難しいが、あらかじめそれぞれのバリエーションの仕様が明らかになっていれば、ソースレベルで両方の仕様に対応できるようなプログラムを書くことは難しくない。例えば、ＰＳＴＬＢ命令の機能を実装しない場合に、ＰＳＴＬＢをＰＴＬＢとして実行するのは①の方針であり、ＥＩＴ（ＲＩＥ９とするのは②の方針となる。①か②かは個別に検討する必要がある。

【０３４７】・ＰＳＴＬＢの／ＳＴオプションについて

ＴＬＢやキャッシュのインプリメント方式によっては、／ＳＴオプションの実装が難しい場合がある。しかし、／ＳＴオプションの指定は、ＴＬＢのパージ範囲だけに影響するので、互換性維持のために①の方針をとる。すなわち、／ＳＴオプションの実装が難しい場合には、ＥＩＴではなく／ＡＴ想到の動作を行なうということにする。

274

・ＬＳＩＤ機能の実装とＰＴＬＢ，ＰＳＴＬＢの／ＳＳオプションについて

ＰＴＬＢ，ＰＳＴＬＢの／ＳＳオプションの実装については、ＴＬＢやキャッシュのタグにＬＳＩＤの値を含めない場合にも、／ＳＳオプションで全パージすることにより、①の方針に合わせることは可能である。しかし、ＬＳＩＤ制御レジスタに値を書き込んだり、それを読みだしたりすることを考えると、オブジェクトレベルでの完全な互換性を達成するためには、ＬＳＩＤ制御レジスタに相当するものを設けておく必要がある。ところが、互換性の確保だけのためにレジスタを一実装するのは無駄が多い。また、ハードウエアだけではなくソフトウエアに関しても、ＯＳの中でＬＳＩＤの操作を行なう部分は、ＬＳＩＤの機能を活かさなければ無駄な処理になってしまう。したがって、ＬＳＩＤの有無については②の方針をとる。それに合わせて、／ＳＳオプションの実装についても②の方針とする。

【０３４８】・ＬＳＩＤ機能の実装とＰＴＬＢ，ＰＳＴＬＢの／ＳＳオプションについて

ＰＴＬＢ，ＰＳＴＬＢの／ＳＳオプションの実装については、ＴＬＢやキャッシュのタグにＬＳＩＤの値を含めない場合にも、／ＳＳオプションで全パージすることにより①の方針に合わせることは可能である。しかし、ＬＳＩＤ制御レジスタに値を書き込んだり、それを読みだしたりすることを考えると、オブジェクトレベルでの完全な互換性を達成するためには、ＬＳＩＤ制御レジスタに相当するものを設けておく必要がある。ところが、互換性の確保だけのためにレジスタを一つ実装するものを設けておく必要がある。ところが、互換性の確保だけのためにレジスタに関しても、ＯＳの中でＬＳＩＤの操作を行なう部分は、ＬＳＩＤの機能を生かさなければ無駄な処理になってしまう。したがって、ＬＳＩＤの有無については②の方針をとる。それに合わせて、／ＳＳオプションの実装についても②の方針とする。具体的なＬＳＩＤ関連の仕様は、次のようになっている。

－ＬＳＩＤの機能の有無と／ＳＳオプションの実装の有無を一対一に対応させる。ＬＳＩＤのあるプロセッサは／ＳＳオプションが利用できるし、ＬＳＩＤのないプロセッサは／ＳＳオプションを利用できない。

－ＬＳＩＤのあるプロセッサは、ＬＳＩＤのないプロセッサに対してオブジェクトレベルで完全上位互換とする。つまり、ＬＳＩＤのないプロセッサのために書かれたＯＳは、ＬＳＩＤのあるプロセッサの上でもそのまま動く。ただし、ＬＳＩＤの機能は生かすことができない。

－ＬＳＩＤのあるプロセッサのために書かれ、ＬＳＩＤの機能を生かすようなＯＳは、ＬＳＩＤのないプロセッサでは動かない場合がある。具体的には、ＬＳＩＤ制御レジスタの際に予約機能例外（ＲＦＥ）が発生したり、／ＳＳオプションを指定した命令を実行した際に予約命

275

令例外（ＲＩＥ）が発生したりすることになる。

【０３４９】また、MMU関連の命令におけるＬＳＩＤ関係の仕様は、次のようになっている。

－ＬＳＩＤのでは、ＬＳＩＤのないときＰＳＴＬＢ／ＡＳ／ＡＴ　@ｕｒａｄｄｒ相当の動作を行なう。また、ＬＳＩＤ実装時は、その機能を生かすため、ＬＤＣＴＸでＴＬＢ，キャッシュのパージは行わない。ＬＳＩＤ実装時は、ＴＬＢやキャッシュのタグ部分にＬＳＩＤまで含まれているので、ＬＤＣＴＸによりＬＳＩＤが変更されると、ＬＤＣＴＸ実行前の論理空間でヒットしていたエントリがＬＤＣＴＸ実行後の新しい論理空間ではヒットしなくなる。ヒットしなくても、新しいエントリのロードに伴ってリプレースされない限りパージはされないので、再びＬＤＣＴＸが実行されて以前の論理空間に戻った場合には、そのエントリがそのまま有効になる。（ＬＤＣＴＸでキャッシュやＴＬＢをパージしていたのでは、ＬＳＩＤの意味がなくなってしまう。）

－ＬＤＡＴＥ／ＰＴでは、ＬＳＩＤのない時ＰＳＴＬＢ／ＡＳ／ＰＴ相当の動作を行なう。ＬＳＩＤ実装時、ｐｒｇａｄｄｒがＳＲであればＰＳＴＬＢ／ＡＳ／ＰＴ相当の動作を行ない、ｐｒｇａｄｄｒがＵＲであればＰＳＴＬＢ／ＳＳ／ＰＴ（ＬＳＩＤ＿ｏｆ＿ＴＡＧ＝Ｒ０）相当の動作を行なう。ｐｒｇａｄｄｒがＳＲかＵＲかで動作が異なっているように見えるが、これはＬＤＡＴＥの問題ではあく、ＰＳＴＬＢの動作がＵＲとＳＲで異なっているためである。ＬＤＡＴＥ命令自体の動作は、ＡＴＥの変更に伴って影響を受けるＴＬＢやキャッシュをパージする、ということで一貫している。

－ＬＤＡＴＥ／ＳＴでは、ＬＳＩＤのない時ＰＳＴＬＢ／ＡＳ／ＰＴ相当の動作を行なう。ＬＳＩＤ実装時、ｐｒｇａｄｄｒがＳＲであればＰＳＴＬＢ／ＡＳ／ＰＴ相当の動作を行ない、ｐｒｇａｄｄｒがＵＲであればＰＳＴＬＢ／ＳＳ／ＳＴ（ＬＳＩＤ＿ｏｆ＿ＴＡＧ＝Ｒ０）相当の動作を行なう。（／ＰＴと同様）

－ＬＤＣ命令でＵＡＴＢ，ＳＡＴＢを変更した場合は、仮想的な命令ＬＤＡＴＥ／ＡＴを実行したと考えれば良く、ＬＤＡＴＥ／ＰＴ，ＬＤＡＴＥ／ＳＴと同様の動作を行なう。すなわち、ＬＳＩＤのない時、ＳＡＴＢを変更した場合にはＰＳＴＬＢ／ＡＳ／ＡＴ（ｐｒｇａｄｄｒ＝ＳＲ）相当の動作を行ない、ＵＡＴＢを変更した場合にはＰＳＴＬＢ／ＡＳ／ＡＴ（ｐｒｇａｄｄｒ＝ＵＲ）相当の動作を行なう。ＬＳＩＤ実装時、ＳＡＴＶを変更した場合にはＰＳＴＬＢ／ＡＳ／ＡＴ（ｐｒｇａｄｄｒ＝ＳＲ）相当の動作を行ない、ＵＡＴＢを変更した場合にはＰＳＴＬＢ／ＳＳ／ＡＴ（ＬＳＩＤ＿ｏｆ＿ＴＡＧ＝ＬＳＩＤ，ｐｒｇａｄｄｒ＝ＵＲ）相当の動作を行なう。

【０３５０】付録４．本発明装置のフラッグ変化

各命令のフラッグ変化の表記法は、以下の通りとする。

276

－　変化なし

＋　本来のフラッグの意味に合わせて変化する

＊　本来のフラッグの意味とは異なった変化をする

０　０にクリアされる

１　１にセットされる

【０３５１】Ａ４－１．データ転送命令

図３１６に示す。

【０３５２】Ａ４－２．比較・テスト命令

図３１７に示す。

【０３５３】Ａ４－３．算術演算命令

図３１８に示す。ＡＤＤＸ，ＳＵＢＸのＸ＿ｆｌａｇは、ｄｅｓｔのサイズからの桁上げ、桁下げを示す。ＳＵＢＸでｓｒｃとｄｅｓｔのサイズが等しい場合、Ｘ＿ｆｌａｇは符号なし演算としての大小関係といった意味にもなる。一方、Ｌ＿ｆｌａｇの方は符号付き演算としての大小関係を表わす。ＭＵＬ，ＭＵＬＵ，ＭＵＬＸ，ＤＩＶ，ＤＩＶＵ，ＤＩＶＸ，ＲＥＭ，ＲＥＭＵ，ＮＥＧのＭ＿ｆｌａｇ，Ｚ＿ｆｌａｇは、オーバーフローの発生にかかわらずｄｅｓｔ設定値が基準である。ＭＵＬＸ，ＤＩＶＸのＭ＿ｆｌａｇ，Ｚ＿ｆｌａｇは、ｒｅｇ設定値には関係しない。ＤＩＶのＶ＿ｆｌａｇは、０除算または（最小負数）÷（－１）の時にセットされる。ＤＩＶＵのＶ＿ｆｌａｇは０除算の時にセットされる。ＤＩＶＸのＶ＿ｆｌａｇは、０除算または商がｄｅｓｔのサイズに入らない時にセットされる。ＮＥＧのＶ＿ｆｌａｇは、ｄｅｓｔが最小負数であった時にセットされる。ＩＮＤＥＸのＭ＿ｆｌａｇ，Ｚ＿ｆｌａｇは、ｘｒｅｇ設定値（結果の一部）が基準である。Ｌ＿ｆｌａｇは結果が負であること、Ｖ＿ｆｌａｇは、乗算または加算でのオーバーフローを示す。

【０３５４】Ａ４－４．論理演算命令

図３１９に示す。ＮＯＴのＭ＿ｆｌａｇ，Ｚ＿ｆｌａｇは、ｄｅｓｔ設定値（反転結果）が基準である。

【０３５５】Ａ４－５．シフト命令

図３２０に示す。Ｍ＿ｆｌａｇ，Ｚ＿ｆｌａｇはｄｅｓｔ設定値（シフト結果）が基準である。Ｘ＿ｆｌａｇは最後にシフトアウトされた値が入る。ＳＨＡ，ＳＨＬ，ＲＯＴでｃｏｕｎｔが０の場合には、Ｘ＿ｆｌａｇ＝０となる。ＳＨＡでは、ｃｏｕｎｔ＞０で符号の変化があった場合にのみＶ＿ｆｌａｇ＝１。それ以外の場合はＶ＿ｆｌａｇ＝０。

【０３５６】Ａ４－６．ビット操作命令

図３２１に示す。

【０３５７】Ａ４－７．固定長ビットフィールド命令

図３２２に示す。固定長ビットフィールド命令では、ＢＦＣＭＰ，ＢＦＣＭＰＵがＣＭＰ，ＣＭＰＵに準じたフラッグ変化をし、それ以外の命令がＭＯＶ，ＭＯＶＵに準じたフラッグ変化をする。ＢＦＩＮＳ，ＢＦＩＮＳＵの場合、図３２３のＢＢＢＢＢＢＢＢがフラッグ変化の

基準となる。また、ＢＦＥＸＴ，ＢＦＥＸＴＵでは、抽出されたビットフィールドではなく、デスティネーションに設定される値を基準としてフラッグ変化する。これは、ＭＯＶ命令などにおいて、デスティネーション側に設定された値を基準としてフラッグ変化するのに合わせたものである。

【０３５８】Ａ４－８．任意長ビットフィールド命令

図３２４に示す。

【０３５９】Ａ４－９．１０進演算命令

図３２５に示す。ＢＣＤ数は符号拡張が無意味なので、基本的に符号なしの数を扱うものと考え、ＡＤＤＵ，ＳＵＢＵに準じたフラッグ変化とする。なお、ＡＤＤＸ，ＳＵＢＸは、符号なし、符号付きの両方の数を扱うため、多少変則的なフラッグ変化になっており、ＡＤＤＵ，ＡＤＤＤＸ，ＳＵＢＵ，ＳＵＢＤＸとは異なったフラッグ変化である。本発明装置では１０進演算はサポートしない。

【０３６０】Ａ４－１０．ストリング命令

図３２６に示す。ＳＭＯＶ，ＳＣＭＰ，ＳＳＣＨのＦ＿ｆｌａｇは、終了条件による終了（ＳＳＣＨの場合はサーチ成功）を示す。Ｖ＿ｆｌａｇは、エレメント数によって命令を終了した場合を示す。Ｍ＿ｆｌａｇは、複数の終了条件を区別するために用いる。Ｒ３に関係する条件で終了した場合には０、それ以外の０またはＲ４の関係で終了した場合（〈〈Ｌ２〉〉のみ）には１となる。ＳＣＭＰのＸ＿ｆｌａｇ，Ｌ＿ｆｌａｇ，Ｚ＿ｆｌａｇは、最終エレメントの比較結果を基準としてセットされる。Ｘ＿ｆｌａｇはエレメントを符号なしデータと考えた時の大小、Ｌ＿ｆｌａｇはエレメントを符号付きデータと考えた時の大小を示す。

【０３６１】Ａ４－１１．キュー操作命令

図３２７に示す。ＱＩＮＳのＺ＿ｆｌａｇは、空のキューに挿入したことを示す。ＱＤＥＬのＺ＿ｆｌａｇは、エントリの削除の結果キューが空になったことを示す。また、ＱＤＥＬのＶ＿ｆｌａｇは、空のキューからエントリを削除しようとしたことを示す。ＱＳＣＨのＦ＿ｆｌａｇは、終了条件による終了（サーチ成功）を示す。Ｖ＿ｆｌａｇは、キュー終了値Ｒ２による終了（サーチ失敗）を示す。Ｍ＿ｆｌａｇは、複数の終了条件を区別するために用いる。Ｒ３に関係する条件で終了した場合には０、それ以外の０またはＲ４の関係で終了した場合（〈〈Ｌ２〉〉のみ）には１となる。

【０３６２】Ａ４－１２．ジャンプ命令

図３２８に示す。ジャンプ関係の命令では、フラッグは全く変化しない。

【０３６３】Ａ４－１３．マルチプロセッサ命令

図３２９に示す。

【０３６４】Ａ４－１４．制御空間、物理空間操作命令

図３３０に示す。ＬＤＣでｄｅｓｔにＰＳＷを指定した場合には、全フラッグが変化する。

【０３６５】Ａ４－１５．ＯＳ関連命令

図３３１に示す。本発明装置ではＪＲＮＧ，ＲＲＮＧはサポートしない。

【０３６６】Ａ４－１６．ＭＭＵ関連命令

図３３２に示す。ＡＣＳ命令のＭ＿ｆｌａｇはｒｅａｄ可、Ｌ＿ｆｌａｇはｅｘｕｃｕｔｅ可、Ｚ＿ｆｌａｇはｗｒｉｔｅ可を示す。ＭＯＶＰＡのＶ＿ｆｌａｇは、ページフォールトまたはエラーにより物理アドレスが得られなかったことを示す。Ｆ＿ｆｌａｇは、ページフォールトを示す。ＬＤＡＴＥ，ＳＴＡＴＥのＶ＿ｆｌａｇは、ページフォールトまたはエラーによりＡＴＥの転送ができなかったことを示す。本発明装置ではＡＣＳ命令以外のＭＭＵ関連命令をサポートしない。

【０３６７】付録５．異種サイズ間の演算について

本発明装置では、異なるバイト数の整数の間で各種の演算を行なうことができ、これを「異種サイズ間の演算」と呼んでいる。「異なるサイズ」といっても、現在は整数のみを対象としているため、データサイズの変換はゼロ拡張または符号拡張のみの簡単な処理で済む。例えば、３２ビットの整数に８ビットの符号付き整数を加える場合、８ビット整数の符号ビット（ＭＳＢ）を上位ビットにまで拡張してから加算を行なうわけである。符号拡張処理はゲート１～２段で可能なので、一般の加算命令と比べてそれほど複雑なわけではない。

【０３６８】Ａ５－１．異種サイズ間の演算の有用性

異種サイズ間の演算は、次のような場合によく使用する。

①オペランドの一方がイミディエート値の時

変数と定数の演算を行なう場合、定数のサイズはコンパイル時にわかるため、定数の方を小さいサイズとして扱えば、命令長を減らすのに効果がある。例えば、３２ビットレジスタに８ビット定数の１００を加える場合、３２ビットの加算命令を使うと定数１００を指定するのにも３２ビットのフィールドが必要である。しかし、３２ビットに８ビットを加算する命令を使えば、定数１００を指定するフィールドが８ビットで済むため、命令が短くなる。さらに、乗除算の場合には、命令長だけではなく性能面でも影響がある。マイクロプロセッサでは３２～６４ビットの並列乗算器を持つのは苦しいので、どうしても加算とシフトで乗算を行なうことになるが、乗算の計算量は、２つのオペランドサイズの積に比例して多くなるため、２つのオペランドのうちの一方だけでもサイズの小さい方が有利である。異種サイズ間演算の機能がない場合には、例えば、３２ビットの変数に３を掛けるだけでも３２ビット＊３２ビットの乗算を行なわなければならない。

【０３６９】②アドレス計算

アドレスの計算では、演算のデスティネーションのサイズをアドレス幅と同じにする必要がある。したがって、３２ビットプロセッサの場合、３２ビットと他のサイズ

279

との演算をよく行なう。例えば、文字の変換テーブルなどにおいて、テーブルのインデクスの範囲が８ビットにおさまる場合、インデクスとベースアドレスとの加算は、８ビット符号なし整数と３２ビット整数との加算として実現される。

③高級言語

一般に、高級言語では、サブルーチンパラメータのサイズを必ずマシンの基本サイズ（例えば３２ビット）に拡張することが多い。これは、スタックを使ってサブルーチンパラメータの受け渡しをするため、また分割コンパイル等を容易にするためである。さらに、言語Ｃのように、式の中の変数のデータサイズにかかわらず、式の評価は必ずマシンの基本サイズで行なうという場合もある。一方、メモリ上の変数、特に配列では、メモリ領域の節約のため、必要最小限のサイズとするのが普通である。したがって、配列とサブルーチンを同時に使用するプログラムでは、データの移動中または演算処理の途中のどこかでサイズの変換を行なわなければならない場合＊

280

＊が出てくる。式の評価とサイズの変換を同時に行なうためには、本発明装置のような異種サイズ間の演算が便利である。

【０３７０】Ａ５－２．本発明装置における実際

本発明装置では、異種サイズ間の演算をサポートするため、データサイズの指定に関する直交性が非常に強くなっており、２オペランド一般形（Ｇ－ｆｏｒｍａｔ）の基本演算命令のほとんどで異種サイズ間の演算が可能である。つまり、２オペランド一般形の基本演算命令では、ソースのサイズとディスティネーションのサイズが独立に指定でき、必要に応じて符号拡張、ゼロ拡張、上位ビットの切り捨て、などを行なうようになっている。デスティネーションのサイズがソースのサイズより小さい場合にも演算は実行され、デスティネーションのサイズに従ってオーバーフローが検出される。

【０３７１】以下に、各命令における異種サイズ間演算の実際例を述べる。

B:　Byte　　8ビット

H:　Halfword　16ビット

W:　Word　　32ビット

MOV　src.B.dest.W

8ビットのsrc を32ビットに符号拡張してdestに転送。

MOV　src.W.dest.B

src の下位8ビットをdestに転送。

32ビット符号付き整数としてのsrc の値と、8ビット符号付き整数としてのdestの値が異なる時は、オーバーフローとなる。

ADD　src.B.dest.W

8ビットのsrc を32ビットに符号拡張してdestに加算。

ADD　src.W.dest.B

destにセットされる値は、src の下位8ビットをdestに加算したものと同じである。しかし、命令の意味としては、src （32ビット）とdest（8ビットを32ビットに符号拡張）を加算し、それを8ビット符号付き整数に変換してdestに格納するということである。したがって、32ビットの和がdestの8ビットで表現できない値になった時は、オーバーフローとなる。

本発明装置では、ソースとデスティネーションのデータサイズが異なる場合に、通常符号拡張を行なう。ただし、特にゼロ拡張も必要と考えられる命令（MOV，CMP，ADD，SUB）については、ゼロ拡張と符号拡張を命令レベルで切り分け、MOVU，CMPU，ADDU，SUBU命令としている。MOVU，CMPU，ADDU，SUBU　（加えてMULU，DIVU）では、デスティネーションのサイズがソースのサイズよりも大きい時にゼロ拡張を行ない、結果を符号なし整数と考えてオーバーフローの検出をする。

281

【０３７２】Ａ５－３．異種サイズ間の論理演算
論理演算の場合は各ビットが全く独立なので、異種サイズ間の演算は意味がないし（フラッグの変化を除けば、小さい方のサイズで行なうのと同等）、論理演算のオペランドに対するゼロ拡張や符号拡張もほとんど意味を持たない。しかし、例えばＣで次のような関数を書いた場合には、意味はなくても
符号拡張→論理演算
というオペレーションを実行しなければならないことがある。

```
                foo( ) {
    short   int16;
            /* 16 ビット符号付き整数 */
    int     int32;
            /* 32 ビット符号付き整数 */

    int32 &= int16;
            /* int16は符号拡張される */

}
```

ただ、このような例は、言語としての規則性や対称性のためにそう決まっているだけであり、一部のトリッキーなプログラムを除けばほとんど使わない機能であると言える。
【０３７３】論理演算の異種サイズ間演算をサポートするかどうかについて、問題点をまとめると以下のようになる。
①実行時
異種サイズ間の論理演算は、頻度としては非常に少なく、論理的な意味も持たない。本質的に他の命令で代用できるか、あるいはトリッキーなプログラムでしか使わない。
②コンパイル時
Ｃ言語では、論理演算でもゼロ拡張や符号拡張が必要になることがある。あまり使わない機能であっても、コンパイラとしては必ず正しいコードを出す必要がある。命令の対称性が重要。
③チップのインプリメント
符号拡張／ゼロ拡張の区別が全命令で統一的に行なわれていれば、インプリメントの規則性の面から、論理演算でもゼロ拡張や符号拡張を導入するメリットがある。しかし、そのためには命令の割り当てに多くのビットパターンが必要となり、命令のエンコーディングが苦しくなってしまう。現実的には、論理演算を含む全命令で符号拡張／ゼロ拡張の区別を行なうことはできず、論理演算の符号拡張やゼロ拡張に対してインプリメントの規則性の面からのメリットは得られない。また、この部分はメーカー間で見解の異なる可能性もあるため、仕様を合わせるのが難しい。

282

【０３７４】結局、以上の中で②と③のどちらを重視するかということになるが、実質的な性能向上をねらうのであれば、やはり③を選択するのが適当と考えている。つまり、
・異種サイズ間の論理演算のように、実行する意味の少ない演算が足を引っ張ることによって、性能向上がはばまれるのはよくない。
・②の問題に関しては、符号拡張を含む異種サイズ間の論理演算は頻度の少ない演算であるから、多少実行速度が落ちてもよい。
例えば、

```
        AND     src.B.dest.W
                        srcを符号拡張
```

の代わりに

```
        MOV     src.B.@-SP.W
                        src を符号拡張
        AND     @SP+.W.dest.W
```

とすれば、多少実行速度は落ちるが、符号拡張～演算命令の統一的な置き換えが可能である。こうすれば、コンパイラの負担はそれほど増えない。したがって、本発明装置の仕様としては、異種サイズ間の論理演算をサポートしていない。異種サイズ間の論理演算に相当する命令ビットパターンを実行した場合には、動作を保証しないということになっている。
【０３７５】Ａ５－４異種サイズ間演算機能のまとめ
本発明装置でサポートする命令と、整数のデータタイプとの関係をまとめると、次のようになる。
・８，１６，３２，６４ビット長の整数をサポートする。
・符号付きの整数を優先してサポートする。
・符号付き整数の算術演算に関しては、２オペランド命令において異種サイズ間の演算がサポートされている。ソースのサイズとディスティネーションのサイズは完全に独立に指定でき、サイズの制限はない。ソースの方がサイズが小さい場合は、符号拡張される。結果は符号付き整数として扱われ、それに従ってフラッグ類がセットされる。
・符号なし整数の演算は、一部の命令（ＭＯＶ，ＣＭＰ，ＡＤＤ，ＳＵＢ，ＭＵＬ，ＤＩＶ）でのみサポートされている。サイズに関してはやはりソースのサイズとディスティネーションのサイズが完全に独立に指定できる。ソースの方がサイズが小さい場合は、ゼロ拡張される。結果は符号なし整数として扱われ、それに従ってフラッグ類がセットされる。
・符号付きの整数と符号なしの整数の混在した演算はできない。ただし、加算命令などの場合は、デスティネーションの符号の有無はフラッグに影響するだけなので、

フラッグを見ないのであればＡＤＤまたはＡＤＤＵで代用できる。

・異種サイズ間の論理演算はサポートしない。

【０３７６】付録６．高級言語向きサブルーチンコール

高級言語におけるサブルーチンコールでは、単なるリターンアドレスの退避だけではなく、フレームポインタの設定、ローカル変数の領域の確保、汎用レジスタの退避といった処理も必要である。これらの処理はＪＳＲ，ＳＴＭなどの命令に分解することも可能であるが、頻度が多いこと、処理が定型的であることにより、一つの命令（ＥＮＴＥＲ，ＥＸＩＴＤ）としてまとめられている。

【０３７７】Ａ６－１．本発明装置でのサブルーチンコール

高級言語（特にＣ，ＰＡＳＣＡＬ）のサブルーチンコールでは、普通図３３３のような処理を行なう。本発明装置では、高級言語用サブルーチン命令ＥＮＴＥＲとリターン命令ＥＸＩＴＤを設けている。いくつか注意する点を述べる。

・ＦＰ（フレームポインタ）とディスプレイレジスタ

ＰＡＳＣＡＬのようにスタティックスコープがある言語では、中間レベル（ローカル変数とグローバル変数の中間のレベル）の変数のアクセスにディスプレイレジスタを用いる。本発明装置のようにレジスタの多いプロセッサでは、汎用レジスタ上にこのディスプレイレジスタを置くのが有効である。これは、複数のＦＰを持つことに相当する。（実現方法は後述）

【０３７８】・パラメータ

パラメータを渡すには、パケットにしてその先頭アドレスをレジスタ等で渡す方法、パラメータをスタックに積む方法、等があり、高級言語では後者の方が一般的である。呼ばれたサブルーチンの側でスタック上のパラメータをアクセスするにはＦＰ相対のモードを使うことが多い。サブルーチン実行後には、呼んだ側でスタック上のパラメータを解放する必要がある。言語によって、また分割コンパイルをしなければ、呼ばれた側で正確なパラメータ数がわかっているので、リターン命令の中で解放するパラメータ数（ＳＰへの加算値）を指定することができる。本発明装置では、この目的でＥＸＩＴＤ命令を設けている。パラメータ数が動的に決まる場合もあるので（例えば、パラメータ数をサブルーチンに知らせるために、特定のレジスタやスタックを使う場合など）、ＥＸＩＴＤ命令におけるＳＰへの加算値としては、イミディエート値だけではなくレジスタ上の値を使用することもできる。しかし、言語Ｃのようにサブルーチンのパラメータの個数が確定しない言語では、サブルーチンを呼ばれた側からは、サブルーチンを呼ぶ側で決めたパラメータの個数がわからない。したがって、呼ばれた側で実行するＥＸＩＴＤ命令では、解放するパラメータの個数を指定できない。その場合には、サブルーチンを呼んだ側でＡＤＤ　＃ｎ，ＳＰを行なってパラメータを解放する。

【０３７９】結局、本発明装置のＥＮＴＥＲ命令では前の図の２．～４．の処理、ＥＸＩＴＤ命令では５．～７．の処理、または５～８の処理（ただし、８．で解放するパラメータ数はサブルーチン側で指定）を行なうことになる。１．はＪＳＲと同じ処理になり、８．はサブルーチンを呼んだ側でＡＤＤ　＊＊＊，ＳＰを行なう。

本発明装置における高級言語でのスタックフレームは、図３３４のようになる。

・ｌｏｃａｌ　ｖａｒｉａｂｌｅｓとｐａｒａｍｅｔｅｒｓがなるべくＦＰから近い配慮となるように、ローカル変数確保よりレジスタセーブを後にした。

・ＥＸＩＴＤ命令にはＰＣのリストア（ＲＴＳ）の操作まで含めている。

命令シーケンスの実際

（サブルーチン側でパラメータ数がわからない場合）図３３５に示す。

（サブルーチン側でパラメータ数がわかる場合）図３３６に示す。

【０３８０】Ａ６－２．ブロック構造言語のためのディスプレイレジスタの構成例

ＥＮＴＥＲ－ＥＸＩＴＤで処理するＦＰレジスタをダイナミックリンクとして利用するためには、内側のブロック（最高のレキシカルレベル）に対するフレームポインタにＦＰレジスタを割り当てるのがよい。その他のレキシカルレベルのフレームポインタには、値の変化の少ない順にＲ１３，Ｒ１２，Ｒ１１．．．を使う。つまり、レキシカルレベルの高い内側のブロックへ行くにしたがって番号の小さいレジスタを使い、最小番号のレジスタの内容をＦＰと同じにする。各サブルーチンでは、ＥＮＴＥＲ命令実行後、ＦＰを自分のレキシカルレベルに対応したフレームポインタ用レジスタにコピーし、その番号以上のレジスタをディスプレイレジスタとして、その番号以下のレジスタを評価用として利用すればよい。ただし、新しく書き換えたレジスタは必ず退避して値を保存しなければならない。

【０３８１】プログラム例（スタティックスコープ）

図３３７に示す。

実行状態の例（ダイナミックリンクとディスプレイレジスタ）

図３３８に示す。

- proc0*. var0*

ｐｒｏｃ０であるが、再帰呼び出しのため、最初のｐｒｏｃ０とは異なったフレームになっている。

・ＦＰのコピーにより破壊するレジスタは、コピー前にすべてＥＮＴＥＲ命令で退避しなければならない。レジスタを退避しておけば、サブルーチンの実行が終って一つ前の関数に戻った時、レキシカルレベルの大小にかかわらずディスプレイはもとの値に戻る。上の例では、レジスタの使い方について次のような関係が成立する。レ

285

キシカルレベルｎのサブルーチン実行に関して
①Ｒ１３．．Ｒ１３－ｎ＋１のｎ個のレジスタはディスプレイレジスタとして参照のみを行ない、書き込みはしない。
②Ｒ１３－ｎのレジスタは、このレベルのローカル変数のディスプレイとして使用するため、ＥＮＴＥＲ実行後ＦＰからコピーする。このディスプレイは、このサブルーチン実行中にさらに高いレベルのサブルーチンを呼んだ時に、呼ばれたサブルーチンからこのレベルの変数をアクセスするために使用する。このサブルーチンからこのレベルの変数をアクセスするには、同じ内容であるＦＰを使うのが良い。
③Ｒ１３－ｎ－１．．Ｒ０の（１３－ｎ）個のレジスタは、レジスタ変数や評価用として利用する。
④Ｒ１３－ｎのレジスタとＲ１３－ｎ－１．．Ｒ０のうちで使用するレジスタは、必ずＥＮＴＥＲ命令で退避する必要がある。全部のレジスタが保存されなければならない。
【０３８２】付録７．制御レジスタと制御空間
制御レジスタ関係の仕様は、チップバス（コプロセッサやキャッシュ、ＴＬＢなどに接続されるバス）やインプリメント方法との関係が深いため、〈〈ＬＡ〉〉仕様となっている。
【０３８３】Ａ７－１．制御空間の考え方
「本発明装置」では、チップバス上のメインプロセッサやコプロセッサに含まれるすべてのレジスタ、MMUやキャッシュ、ＴＬＢなどの制御レジスタ、およびチップバス上に接続されたコンテキストスイッチ用高速メモリに一意的なアドレスを付け、これを制御空間と呼ぶ。
「本発明装置」の制御空間は、従来のプロセッサに見られたコプロセッサ用のアドレス空間（Ｃｏｐｒｏｃｅｓｓｏｒ－ＩＤなど）をメインプロセッサの制御レジスタのアドレスと統合化、一般化したものであり、次のような特徴を持つ。
・「本発明装置」の制御空間には次のようなものが含まれる。
メインプロセッサの制御レジスタ
ＰＳＷ，各リングのスタックポインタなど
MMU関係の制御レジスタ（「本発明装置」はMMUを内蔵しない）
ＵＡＴＢ，ＳＡＴＢなど
インプリメントに依存して必要となるレジスタ
［コプロセッサの制御レジスタ］
［コンテキスト退避用の高速メモリ］将来チップに内蔵することを狙ったもの
［プロセッサ内の汎用レジスタ、テンポラリレジスタ］
外部からの診断、デバッグの目的
・制御空間は、コンテキスト（プロセスやタスク）間で共通の空間である。制御空間のアクセスはアドレス変換を伴わないため、単純化されたプロトコルで高速のアクセスが可能である。この機能は、特に高速コンテキストスイッチでも利用される。制御空間の考え方は、将来、コプロセッサやコンテキスト退避用メモリがメインプロセッサに内蔵された時に、特に有効になると考えられる。しかし、最初のバージョンのチップでは、インプリメントの制約上、あるいはチップバスの構成上、制御空間の操作を統一的に行なうのが難しい場合があるので、将来に備えてアドレスの割り当てのみを決めておき、制御空間操作命令はいくつかの制限を付けて使用してもよいということにする。

286

【０３８４】具体的には、次のような制限が付く。
・プロセッサの診断用の目的で、Ｒ０～Ｒ１５やＰＣにも制御空間のアドレスが割り当てられているが、これは〈〈Ｌ２〉〉であり、「本発明装置」では実装しない。
・ＬＤＣ，ＳＴＣは、本来メインプロセッサ内の制御レジスタ、ＦＰＵの制御レジスタ、コンテキスト退避用メモリなどを統一的にアクセスすることを狙ったものである。しかし、「本発明装置」ではインプリメントの制限により、メインプロセッサ内の制御レジスタ以外（実効アドレスH’０～H’０７ｆｆ以外）のものについては、ＬＤＣ，ＳＴＣでアクセスできない。
・「本発明装置」の制御空間のアドレスでは、バイトアクセス、ハーフワードアクセスが利用できない。また、ワードアクセスにおいてアラインメントが強制される。
・コンテキスト退避用メモリは、制御レジスタを置く領域（H’０～）と重ねることはできない。コンテキスト退避用メモリとしては、H’ｆｆｆｆ８０００～H’ｆｆｆｆｆｆｆｆのアドレスが割り当てられている（さらに拡張領域としてH’８０００００００～）ので、H’８０００００００～H’ｆｆｆｆｆｆｆｆ以外の値をＣＴＸＢＢに設定してＬＤＣＴＸ／ＣＳ，ＳＴＣＴＸ／ＣＳを実行した場合には、エラーとする。また、ＬＤＣＴＸ／ＣＳ，ＳＴＣＴＸ／ＣＳの機能自体も〈〈Ｌ２〉〉となっている。
【０３８５】・「本発明装置」ではＬＤＣＴＸ／ＣＳ，ＳＴＣＴＸ／ＣＳをサポートしない。

—— ：必須の仕様<<L1>>
…… ：アドレス割り当てのみ<<L2>>

図３３９に示す制御空間では、バイトアクセス、ハーフワードアクセスができないのにもかかわらず、バイトアドレッシングとなっている。これは、一般の命令で利用される論理空間と同じように、制御空間でも汎用アドレッシングによる実行アドレス指定が可能であり、論理空間と同じ形のバイトアドレッシングにしておかないと、混乱を招くからである。また、制御空間で汎用アドレッシングを利用可能としたのは、制御空間をコンテキスト退避に利用することを考えたためである。ＬＤＣ，ＳＴＣでメインプロセッサ内の制御レジスタしかアクセスできない場合には、バイトアドレッシングとした意味がな

287

くなってしまい、多少不自然な仕様になる。しかし、その背景には上記のような将来的な見通しがあり、一部の機能のみを実現した場合に多少不自然に見えるのはやむを得ない。

【０３８６】Ａ７－２．メインプロセッサの制御レジス＊

H'0000～H'03ff　メインプロセッサ、
H'0400～H'07ff　メインプロセッサ、
H'0800～H'0bff　FPU(TRON reserve)
H'0c00～H'0fff　FPU<<LV>>

...

＊はコンテキスト毎に別々に用意されるレジスタ
／は必ずしも実装（アドレス割り当て）をしなくても良いレジスタ

【０３８７】

アドレス		レジスタ
H'0000		reserved
H'0004	＊	PSW
H'0008		reserved
H'000c	(＊)	SMRNG
H'0010		reserved
H'0014	(＊)	IMASK
H'0018		reserved
H'001c		reserved
H'0020		reserved -- EITVBH
H'0024		EITVB
H'0028		reserved -- JRNGVBH

288

＊タ

制御レジスタのニモニックとアドレスについては、次のようになる。制御レジスタのアドレスが８ｎ＋４の位置にあるのは、レジスタを６４ビットに拡張することを考慮したためである。

MMU(TRON reserve)
MMU <<LV>>

H'002c		「本発明装置」
		reserved -- JRNGVB
H'0030		reserved -- CTXBBH
H'0034	＊	CTXBB
H'0038		reserved
H'003c		reserved
H'0040		reserved -- SATBH
H'0044		「本発明装置」
		reserved -- SATB
H'0048		reserved -- UATBH
H'004c	＊	「本発明装置」
		reserved -- UATB
H'0050		reserved
H'0054	＊	「本発明装置」
		reserved -- LSID
H'0058		reserved
H'005c		reserved
H'0060		reserved -- IOADDRH
H'0064	／	IOADDR
H'0068		reserved -- IOMASKH
H'006c	／	IOMASK
H'0060～H'007f		reserved

【０３８８】

H'0080		reserved
H'0084	(＊)	「本発明装置」
		reserved -- DCB
H'0088		reserved
H'008c		DI
H'0090		reserved

289

H'0094	* 「本発明装置」
	reserved — CSW
H'0098	reserved
H'009c	(*)「本発明装置」
	reserved -- CTXBFM
H'00a0～H'00ff	reserved

【０３８９】

H'0100	reserved -- SPIH
H'0104	SPI
H'0108～H'011f	reserved
H'0120	reserved -- SPOH
H'0124	* SPO
H'0128	reserved -- SP1H
H'012c	* 「本発明装置」
	reserved -- SP1
H'0130	reserved -- SP2H
H'0134	* 「本発明装置」
	reserved -- SP2
H'0138	reserved -- SP3H
H'013c	* SP3
H'0140～H'017f	reserved

【０３９０】

H'0180	reserved -- R0H
H'0184	* 「本発明装置」
	reserved -- R0
H'0188	reserved -- R1H
H'018c	* 「本発明装置」
	reserved -- R1
...	...
H'01e0	reserved -- R12H
H'01e4	* 「本発明装置」
	reserved -- R12
H'01e8	reserved -- R13H
H'01ec	* 「本発明装置」
	reserved -- R13
H'01f0	reserved -- R14H
H'01f4	* 「本発明装置」
	reserved -- R14
H'01f8	reserved -- PCH
H'01fc	* 「本発明装置」
	reserved -- PC

290

【０３９１】

H'0200～H'03ff	reserved
(H'0400 ～H'07ff	<<LV>>)
H'0410	BBC
H'0414	BBP
H'0500	DBC
H'0520	XBP0
H'0524	XBP1
H'0540	OBP0
H'0544	OBP1

【０３９２】Ａ７－３．制御レジスタの未使用ビット制御レジスタのうち、使用していないビットについては、１を書き込んだ場合にそれをチェックしてＥＩＴとするのが望ましい。しかし、中途半端なチェックを行なうと、互換性（特に下位チップとの互換性）を保つのが難しくなること、チェックのためにオーバーヘッドが生じること、により、ＰＳＷを除いて未使用ビットのチェックは行なわないことにする。ＣＴＸＢＦＭなど〈〈Ｌ２〉〉の機能を持つレジスタについても、〈〈Ｌ２〉〉が実装されていない場合にはエラーチェックが行なわれないし、書き込んだ値がそのまま読み出されるとは限らない。ただし、チェックが行なわれていなくても、空きビットには必ず’０’を入れてもらうように、マニュアル等で指導する必要がある。ＰＳＷについては、未使用のビット’－’に’１’を書き込もうとした場合に、予約機能例外（ＲＦＥ）とする。

【０３９３】以下の制御レジスタの内容の説明における’－’，’＝’，’＊’のビットの意味は次のとおりである。

291　　292

'－'　０にreserved（違反時例外発生）

'＋'　１にreserved（違反時例外発生）

このビットに0(1)を書き込むことは可能であるが、このビットに1(0)を書き込もうとするとLDC.LDCTX などで予約機能例外(RFE) を発生する。

'＝'　０にreserved（違反時も無視）

'#'　１にreserved（違反時も無視）

このビットは0(1)を書き込むべきビットであるが、このビットに1(0)を書き込もうとしても単に無視され、このビットに0(1)を書き込んでも1(0)を書き込んでも動作は同じである。

ユーザへのマニュアルには、将来の拡張のためこのビットに0(1)を書き込むように明記する。

'＊'　書き込み時の値は自由。

ハードウエアの動作は'＝','#' と同じであり、どのような値が書き込まれても単に無視する。ただし、このビットは'＝','#' とは異なり、将来チップの機能が拡張された場合にも使用しないビットである。したがって、ユーザが自由な値を書き込んでも構わない。ユーザへのマニュアルの中でも、このビットは無視されるということを明記し、ビットのマスク処理などは省いてもらうようにする。

・ＩＭＡＳＫ，ＳＭＲＮＧ，ＤＩ，ＤＣＥ，ＣＴＸＢＦＭでは、未使用のビットが’＊’になっている。ＰＳＷでは、未使用のビットが’－’になっている。それ以外の制御レジスタでは、未使用のビットは　’＝’である。

・ＰＳＢ，ＰＳＭの未使用フィールドも’－’である。したがって、ＬＤＰＳＢ，ＬＤＰＳＭでも予約機能例外（ＲＦＥ）が発生する。

・’－’のビットの読みだした場合には、’０’が読み出される。’＝’，’＊’のビットを読みだした場合に得られる値は不定である。前に書き込んだ値がそのまま読み出されるとも限らない。

【０３９４】Ａ７－４．制御レジスタの内容

ＰＳＷ

図３４０に示す

Ｐｒｏｃｅｓｓｏｒ　Ｓｔａｔｕｓ　Ｗｏｒｄ

内容については本文を参照。

ＰＳＭ，ＰＳＢ

ＰＳＷのうち、ユーザのアクセス可能な下位の２バイトのみを抜き出したもの。ＬＤＰＳＢ，ＬＤＰＳＭ，ＳＴＰＳＢ，ＳＴＰＳＭ命令によってアクセスする。制御レジスタのうち、ＰＳＢ，ＰＳＭのみがｒｉｎｇ０以外からアクセスできる。

ＩＭＡＳＫ

図３４１に示す

ＰＳＷのうち、個別にアクセスすることの多いＩＭＡＳＫのフィールドを抜き出して別レジスタとしたものである。ＩＭＡＳＫの操作を容易にすることと、性能向上を狙ったものである。ＩＭＡＳＫ以外のフィールドには、何を書き込んでも単に無視される。

ＳＭＲＮＧ

図３４２に示す

ＰＳＷのうち、個別にアクセスすることの多いＳＭＲＮＧのフィールドを抜き出して別レジスタとしたものである。ＳＭ，ＲＮＧの操作を容易にすることと、性能向上を狙ったものである。ＳＭＲＮＧ以外のフィールドには、何を書き込んでも単に無視される。

【０３９５】ＣＴＸＢＢ

図３４３に示す

Ｃｏｎｔｅｘｔ　Ｂｌｏｃｋ　Ｂａｓｅ

ＣＴＸＢのベースアドレスを指すレジスタ。ＬＤＣＴＸ，ＳＴＣＴＸ命令で使用される。ＣＴＸＢＢは「本発明装置」６４への拡張を考え、「本発明装置」３２でも８バイトのアラインメントを強制している。したがって、ＣＴＸＢＢの下位３ビットは’＝＝＝’となる。つまり、０にｒｅｓｅｒｖｅｄであるが、違反時にも無視される。

ＤＩ（Ｄｅｌａｙｅｄ　Ｉｎｔｅｒｒｕｐｔ）

図３４４に示す

293　　294

DI(Delayed Interrupt)要求を示すレジスタ。

DI=0000　優先度０の外部割り込み(NMI) 処理後のDI要求

DI=0001　優先度１の外部割り込み処理後のDI要求

DI=0010　優先度２の外部割り込み処理後のDI要求

...

DI=1110　優先度15の外部割り込み処理後のDI要求

DI=1111　DI 要求なし

【０３９６】ＤＩ（Ｄｅｌａｙｅｄ　Ｉｎｔｅｒｒｕｐｔ）は、ソフトウエアによって外部割り込みを発生させるメカニズムであり、非同期に発生する各種の処理要求をペンディングとしたい時や処理順序をシリアライズしたい時に有効である。優先度の高い外部割り込みの処理が終わった後で別に起動したい処理がある場合、その要求をＤＩに登録しておくことによって自動的に処理が起動される。ＤＩは、外部割り込みに対してＤＣＥと同等の処理を行なうものである。ＲＥＩＴ命令などによってＰＳＷのＩＭＡＳＫが変化した場合に、ＤＩ＜ＩＭＡＳＫであればＤＩのＥＩＴ処理が起動される。このレジスタのＤＩ以外のフィールドに何を書き込んでも単に無視される。

【０３９７】ＣＳＷ

図３４５に示す

Ｃｏｎｔｅｘｔ　Ｓｔａｔｕｓ　Ｗｏｒｄ

このレジスタは、コンテキスト毎に切り換えの必要な情報のうち、ネストの行なわれないものを集めたものである。ＤＣＥ（Ｄｅｌａｙｅｄ　ＣｏｎｔｅｘｔＥｘｃｅｐｔｉｏｎ）要求を示すＤＣＥフィールドと、ＣＴＸＢのフォーマットを示すＣＴＸＢＦＭのフィールドから成る。ＣＴＸＢＦＭの機能は付録８を参照のこと。ＣＴＸＢＦＭの機能がインプリメントされない場合には、ＤＣＥレジスタとＣＳＷレジスタが全く同じ情報を扱うことになるので、ＣＳＷレジスタもインプリメントされない（アクセス時ＲＦＥ）場合がある。この時、ＣＴＸＢに置かれるのは、形式的にはＣＳＷレジスタであるが、実際はＤＣＥレジスタとなる。ＣＳＷとＤＣＥ，ＣＴＸＢＦＭの関係は、ＰＳＷとＩＭＡＳＫ，ＳＭＲＮＧの関係に同じである。ＣＴＸＢに置かれるのは、ＤＣＥ，ＣＴＸＢＦＭなどの情報を圧縮したＣＳＷの方である。「本発明装置」ではＤＣＥ＝’１１１’に固定とする。

【０３９８】ＤＣＥ

図３４６に示す

Ｄｅｌａｙｅｄ　Ｃｏｎｔｅｘｔ　Ｅｘｃｅｐｔｉｏｎ

ＣＳＷのうち、個別にアクセスすることの多いＤＣＥのフィールドを抜き出して別レジスタとしたものである。ＤＣＥの操作を容易にすることと、性能向上を狙ったものである。ＤＣＥ以外のフィールドには、何を書き込んでも単に無視される。ＣＳＷレジスタがインプリメントされない場合、コンテキストスイッチ時にＣＳＷレジスタの代わりにＣＴＸＢとの転送が行なわれるのは、このＤＣＥレジスタである。この場合、コンテキスト退避の時は、’＊’のビットがすべて０となってＣＴＸＢに書き込まれる。また、コンテキストをロードする時は、’＊’の部分のビットの値はチェックされない。

【０３９９】ＣＴＸＢＦＭ

図３４７に示す

Ｃｏｎｔｅｘｔ　Ｂｌｏｃｋ　Ｆｏｒｍａｔ

ＣＳＷのうち、個別にアクセスすることの多いＣＴＸＢＦＭのフィールドを抜き出して別レジスタとしたものである。ＣＴＸＢＦＭの操作を容易にすることと、性能向上を狙ったものである。ＣＴＸＢＦＭ以外のフィールドには、何を書き込んでも単に無視される。このレジスタは〈〈Ｌ２〉〉である。

ＥＩＴＶＢ

図３４８に示す

ＥＩＴ　Ｖｅｃｔｏｒ　Ｂａｓｅ

ＥＩＴ　（例外、割り込み）ベクトルテーブルの先頭の物理アドレスを示す。ＥＩＴＶＢは　「本発明装置」６４への拡張を考え、「本発明装置」３２でも８バイトのアラインメントを強制している。したがって、ＥＩＴＶＢの下位３ビットは’＝＝＝’となる。つまり、０にｒｅｓｅｒｖｅｄであるが、違反時にも無視される。

【０４００】ＪＲＮＧＶＢ

図３４９に示す

ＪＲＮＧ　Ｖｅｃｔｏｒ　Ｂａｓｅ

ＪＲＮＧ命令のベクトルテーブルの先頭の論理アドレスを示す。ＪＲＮＧＶＢによるテーブルのベースアドレスは、「本発明装置」６４への拡張を考え、「本発明装置」３２でも８バイトのアラインメントを強制している。また、ＪＲＮＧＶＢのＬＳＢはＥｎａｂｌｅビットとなっており、Ｅが０の時にはＪＲＮＧ実行禁止となる。したがって、ＪＲＮＧＶＢの下位３ビットは’＝＝Ｅ’で表記されている。’＝’のビットは、０にｒｅｓｅｒｖｅｄであるが、違反時にも無視される。

ＳＰ０～ＳＰ３

図３５０に示す

Ｓｔａｃｋ　Ｐｏｉｎｔｅｒ　ｆｏｒ　ｒｉｎｇ３

ｒｉｎｇ０～ｒｉｎｇ３で使用するスタックポインタ。ＳＰＩ，ＳＰ０～ＳＰ３についてはアラインメントの制約はなく、ＬＳＢまで有効である。

295

ＳＰＩ
図３５１に示す
Ｓｔａｃｋ　Ｐｏｉｎｔｅｒ　ｆｏｒ　Ｉｎｔｅｒｒｕｐｔ
外部割り込み用のスタックポインタ。
【０４０１】ＩＯＡＤＤＲ，ＩＯＭＡＳＫ
図３５２に示す
アドレス変換を行なわない場合（ＰＳＷのＡＴ＝００，１０）において、Ｉ／Ｏ領域の物理アドレスを指定するレジスタである。通常MMUでアドレス変換を行なう場合には、ＰＴＥ中のＮＣビットによってＩ／Ｏ領域の指定を行なうが、システムスタート時などでアドレス変換ができない場合には、ＩＯＡＤＤＲ，ＩＯＭＡＳＫの２つのレジスタを使ってＩ／Ｏ領域の指定を行なう。アドレス変換なしでメモリアクセスをする場合、物理アドレスとＩＯＭＡＳＫの論理積がＩＯＡＤＤＲと等しければ、そこはＩ／Ｏ領域と見なされる。その領域のデータについては、データのキャッシュへの取り込みやプリフェッチが行なわれず、命令の要求するメモリアクセスと実際の物理的なメモリアクセスが１対１に対応する。アドレス変換のある場合には、ＩＯＡＤＤＲ，ＩＯＭＡＳＫレジスタは使用しない。また、プロセッサのインプリメントによってキャッシュやプリフェッチを行なわない場合には、必ずしもＩＯＡＤＤＲ，ＩＯＭＡＳＫレジスタを使用する必要はない。
【０４０２】ＵＡＴＢ
図３５３に示す
Ｕｎｓｈａｒｅｄ　ｒｅｇｉｏｎ　Ａｄｄｒｅｓｓ　ＴｒａｎｓｌａｔｉｏｎＢａｓｅ
内容については付録３を参照。
ＳＡＴＢ
図３５４に示す
Ｓｈａｒｅｄ　ｒｅｇｉｏｎ　Ａｄｄｒｅｓｓ　Ｔｒａｎｓｌａｔｉｏｎ　Ｂａｓｅ
内容については付録３を参照。
ＬＳＩＤ　（「本発明装置」では実装しない）
図３５５に示す
Ｌｏｇｉｃａｌ　Ｓｐａｃｅ　ＩＤ
複数の論理空間の間の区別を行なう番号を入れる。複数の論理空間に属するＴＬＢや論理キャッシュなどを混在させる時に、この番号を利用する。ＬＳＩＤの有効なビット数については、インプリメント依存である。
【０４０３】付録８．「本発明装置」のＣＴＸＢ
Ａ８－１．ＣＴＸＢについて
「本発明装置」はMMUをもたないため「本発明装置」でサポートするＣＴＸＢフォーマットをどのようにするか現在検討中である。ＯＳが、タスク、プロセスなどといった並行処理やコルーチンの機能をサポートしている場合、ＰＣや汎用レジスタなどのハードウエア資源の情報は、並行処理の単位となるそれぞれのプログラム毎に

296

別々に持つのが普通である。これらのハードウエア資源は、プロセッサと同様に時分割で使用されるため、現在実行中でないプログラムに対するハードウエア資源の情報は、メモリなどに退避しておく必要がある。「本発明装置」では、こういった並行処理の単位となるプログラムの流れをコンテキストと呼ぶ。また、それぞれのコンテキストを実行するために、ハードウエア資源の情報をメモリ上にまとめて退避したものをＣｏｎｔｅｘｔ　Ｂｌｏｃｋ（ＣＴＸＢ）と呼ぶ。ＣＴＸＢ自体のおかれる空間は、ＬＤＣＴＸ，ＳＴＣＴＸ命令のオプションとして、論理空間ＬＳ、制御空間ＣＳより選択できる。ＯＳの書きやすさという点ではＬＳを利用するのが適しているが、コンテキストスイッチを特に高速化したい場合や、コンテキストスイッチ退避用のメモリをチップ内に設ける場合には、ＣＳを利用することもできる。ただし、ＣＳの指定は、将来コンテキスト退避用のメモリをチップに内蔵した場合に有効に活用されるものであり、現在は〈〈Ｌ２〉〉となっている。また、「本発明装置」では、現在実行中のコンテキストのＣＴＸＢの先頭アドレスを保持するレジスタが設けられており、これをＣＴＸＢＢａｓｅ　Ｒｅｇｉｓｔｅｒ（ＣＴＸＢＢ）と呼んでいる。「本発明装置」のＣＴＸＢのフォーマットは、次のようになっている。このうちの一部は、ＬＤＣＴＸ，ＳＴＣＴＸ命令により、ハードウエアでサポートされている。
【０４０４】「本発明装置」３２での標準ＣＴＸＢフォーマット
図３５６に示す
一般に、コンテキストスイッチによって切り換える必要があるのは、ＯＳのＰＣや　ＰＳＷではなく、ユーザプログラムのＰＣやＰＳＷである。ところが、ユーザプログラムのＰＣやＰＳＷは、普通はＯＳ呼び出し時にスタック中に退避されている。上記のＣＴＸＢフォーマットにおいて、ＰＣ，ＰＳＷがスタック中に置かれているのは、そのためである。ＳＰＩを使用した外部割り込みの処理ハンドラの最後で直接コンテキストスイッチを行なう場合は、上記のＣＴＸＢフォーマットを実現するために、ＰＣ，ＰＳＷを別命令で転送する必要がある。しかし、この場合はＤＣＥ，ＤＩを活用し、外部割り込みから抜ける時にＤＣＥ，ＤＩを使ってコンテキストスイッチを行なうという方法がある。そうすれば、ＤＣＥやＤＩでＳＰ０を指定することにより、上記のデータ構造が自然に実現できる。
【０４０５】Ａ８－２．ＣＴＸＢの可変性
ＣＴＸＢに含まれる情報のうち、’＊１’～’＊５’の付いた部分は、システム構成などによって可変性のある部分である。これらの点について説明する。ＣＴＸＢの内容やフォーマットは、以下のような要因によって動的に（あるいはコンテキスト毎に）変化することがある。
・ＯＳの構成やMMUの有無（＊１～＊３）

297

ＯＳの構成によっては、コンテキストスイッチでＳＰ１～ＳＰ３の切り換えを行なっても意味のないケースが考えられるため、ＳＰＩ～ＳＰ３を退避したくない場合がある。また、MMUを使用しない用途でのコンテキストスイッチでは、ＵＡＴＢ，ＬＳＩＤを切り換える必要がない。

（＊１）ＪＲＮＧ～ＲＲＮＧでは、外側のリングが内側のリングのスタックに退避されるため、現在リングよりも外側のリングのＳＰの値は意味を持たない。特に、ｒｉｎｇ０でのみ実行されるコンテキストスイッチの時点では、ＳＰ１～ＳＰ３の値が意味を持たない。ＳＰ１～ＳＰ３は、直接あるいは間接にＳＰ０のスタックに保持されているため、ＳＰ０の切り換えによってＳＰ１～ＳＰ３も間接的に切り換わることになるからである。一方、ＴＲＡＰＡ～ＲＥＩＴの中でコンテキストスイッチが起こった場合にはＳＰ１～ＳＰ３の切り換えも必要である。したがって、ＳＰ１～ＳＰ３をＣＴＸＢに含めたい場合と、そうでない場合がある。

（＊２）MMUを実装しない。〈〈Ｌ１Ｒ〉〉仕様では、ＵＡＴＢは不要である。

（＊３）ＬＳＩＤは、複数の論理空間を区別するための番号である。ＬＳＩＤの実装は〈〈Ｌ２〉〉なので、ＬＳＩＤがＣＴＸＢに含まれる場合と、そうでない場合がある。

【０４０６】・退避する汎用レジスタの指定（＊４）

298

コンテキストで使用しないレジスタやＯＳで使用するワーキングレジスタについて、ＣＴＸＢとの退避、復帰を行なわなければ、無駄な転送がなくなり、コンテキストスイッチ時間が短縮される。

・コプロセッサ使用の有無（＊５）

ＦＰＵのレジスタは汎用レジスタとは別になるが、これもコンテキスト情報として持つ必要がある。したがって、そのコンテキストがＦＰＵを使用しているかどうかによって、ＣＴＸＢが動的に変わるケースが生じる。

【０４０７】以上のようなＣＴＸＢのバリエーションに対処するため、「本発明装置」では、以下のような方法をとっている。

・最初のバージョンの〈〈Ｌ１〉〉のチップでは、ＬＤＣＴＸ，ＳＴＣＴＸでＣＳＷ，ＳＰ０～ＳＰ３，ＵＡＴＢの転送のみを行なう。Ｒ０～Ｒ１４については、別命令ＬＤＭ，ＳＴＭを利用して転送し、（＊４）に対処する。

・それ以外のＣＴＸＢのバリエーションに対しては、現在のＣＴＸＢのフォーマットを識別するレジスタ（ＣＴＸＢＦＭ）を導入し、このレジスタによって、ＣＴＸＢに何が含まれ、ＬＤＣＴＸ，ＳＴＣＴＸで何を転送しなければならないかを知ることにする。なお、ＣＴＸＢＦＭとＤＣＥの情報を合わせたものは、ＣＳＷレジスタとしても扱われる。

【０４０８】［ＣＴＸＢＦＭ］図３５７に示す

```
FR      FPU レジスタを退避することを指定
        「本発明装置」標準のFPU レジスタのコンテキスト退避
        を指定する。この機能は、特に、将来FPU がチップに内
        蔵された場合に利用する。
RG      R0-R14を退避することを指定
        この機能は、特に、将来コンテキスト退避用メモリがチ
        ップに内蔵された場合に利用する。
SP      SPの退避を指定
        SP=00  SP0, SP1, SP2, SP3を退避する
        SP=01  reserved
        SP=10  SP0, SP3を退避する（<<L1R>> 用）
        SP=11  SP0のみ退避する
        この機能は、JRNGによりOSを呼び出す場合に、SP1 ～SP
        3 の無駄な転送を行なわないために利用する。
```

299　　300

また、<<L1R>> でSP1.SP2 を実装しない場合に利用する

。

MM　　MMU 関連レジスタの退避を指定

MM=00　UATB を退避

MM=01　UATB.LSIDを退避

MM=10　MMU関連レジスタは退避しない（<<L1R>> 用）

MM=11　reserved

［ただし、ＣＴＸＢＦＭの詳細は検討中である。］

【０４０９】〈〈Ｌ１〉〉で標準的なフォーマットのＣＴＸＢでは、ＬＤＣＴＸ，ＳＴＣＴＸにより、ＣＳＷ（ＤＣＥ，ＣＴＸＢＦＭ）、ＳＰ０～ＳＰ３，ＵＡＴＢが転送される。これは、ＣＴＸＢＦＭをすべて０にすることによって指定される。ＬＤＣＴＸ命令では、ＣＴＸＢからフェッチした新しいコンテキストのＣＳＷの中のＣＴＸＢＦＭを見て、ＣＴＸＢの以下の部分のフォーマットを判断し、指定されたものをロードする。また、ＳＴＣＴＸ命令では、現在のＣＴＸＢＦＭの値を見て、指定されたものをＣＴＸＢに退避する。ただし、ＣＴＸＢＦＭの機能は〈〈Ｌ２〉〉となっており、将来の拡張である。つまり、ＣＴＸＢ固定の仕様が〈〈Ｌ１〉〉，ＣＴＸＢ可変の仕様（上位コンパチ）が〈〈Ｌ２〉〉である。〈〈Ｌ１Ｒ〉〉のチップについては、ＳＰ１，ＳＰ２，ＵＡＴＢの転送が不要なので、ＣＴＸＢにもこれらの値は含めない。ＣＴＸＢにこれらのレジスタ値が含まれているかどうかの識別は、ＣＴＸＢＦＭによって行なうことが可能である。ただし、ＣＴＸＢＦＭの実装が重い場合には、ＬＤＣＴＸ，ＳＴＣＴＸ命令の追加オプションにより直接ＣＴＸＢのフォーマットを指定する仕様や、ＬＤＣＴＸ，ＳＴＣＴＸ命令の追加オプションによりＣＴＸＢＦＭの有効性の有無を指定する仕様にすることも考えられる。

【０４１０】Ａ８－３．ソフトウエアコンテキスト

プロセスやタスク毎に持つ情報の中には、ＯＳがソフトウエアによって管理する情報も含まれている。これらの情報はＯＳによって一定していないので、当然ハードウエア（ＬＴＣＴＸ，ＳＴＣＴＸ命令）ではサポートできない。これらの情報をソフトウエアコンテキストと呼ぶ。例えば、ＩＴＲＯＮの場合、タスク状態、終了時処理ルーチンのアドレス、例外処理ハンドラのアドレス、ｗａｋｅｕｐのカウント、キューの構成のためのリンク用領域などがソフトウエアコンテキストに含まれる。ＣＴＸＢを論理空間（ＬＳ）に置いた場合には、汎用レジスタなどのハードウエアコンテキストとソフトウエアコンテキストを同じように扱うことができる。しかし、ハードウエアコンテキストとして、ＣＳなどの別空間を使用した場合には、ソフトウエアコンテキストもＣＳに置くか（この場合は、ＬＤＣ，ＳＴＣなどの命令が有効である）、あるいは両者をポインタでつないでおいて間接参照するか、といった方法をとる必要がある。

【０４１１】付録９．　「本発明装置」のＥＩＴ処理

ＥＩＴ処理の概要は以下のとおりであるが、細部の仕様は検討中である。通常のプログラム実行の流れをハードウェア機構によって中断し、それに割り込むような形で非同期に起動される処理を、「本発明装置」ではＥＩＴ処理と呼ぶ。ＥＩＴ処理には、次のようなものが含まれる。

・内部割り込み（トラップ－ｔｒａｐ）

・例外割り込み（例外－ｅｘｃｅｐｔｉｏｎ）

・外部割り込み（割り込み－ｉｎｔｅｒｒｕｐｔ）

トラップ、例外、割り込みの区別は、プログラマから見たそのＥＩＴの発生原因によって行なわれるものであり、インプリメント上のメカニズムの違い（スタックに退避される情報の違いなど）を意味するものではない。プロセッサが命令実行中にＥＩＴを検出すると、シーケンシャルな命令の実行を中断してＥＩＴ処理を開始する。ＥＩＴ処理では、ＥＩＴを検出した時のプロセッサの状態をスタックに退避し、ＥＩＴハンドラを起動する。ここまでがプロセッサのハードウエアによって行なわれる。一方、起動されたＥＩＴ処理ハンドラでは、ＥＩＴに応じてエラーの回復、エラーメッセージの表示、エミュレーションなどの処理を行なう。ＥＩＴ処理ハンドラは、ソフトウエアにより実現されるものである。大部分のＥＩＴでは、ＥＩＴ処理ハンドラの最後でＲＥＩＴ命令を発行することにより、中断されたもとの命令列に復帰し、処理を再開することが可能となっている。

「本発明装置」では、将来の機能拡張を考慮し、未定義の命令、不当な命令についてのエラー検出やエミュレーション用のメカニズムを強化する方針である。したがって、命令フォーマットの組み合わせにより不当なオペレーションになる場合や、インプリメントされていない機能を実行しようとした場合には、できるだけエラーとして検出し、例外割り込みを発生する。

【０４１２】Ａ９－１．ＥＩＴの種類

「本発明装置」で発生するＥＩＴには、次のようなものがある。

［メモリ、アドレス関係］

ページ不在例外（ＰＯＥ）・・・・・・・　「本発明装置」では発生しない

Ｐａｇｅ　Ｏｕｔ　Ｅｘｃｅｐｔｉｏｎ（ＰＯＥ）

301

命令またはオペランドアクセス時のアドレス変換において、ＵＡＴＢ，ＳＡＴＢ，ＳＴＥ，ＰＴＥのＰＩビットが０であった場合に発生する。意味的には、ページ不在、ページテーブル不在、セクションテーブル不在を含めたものである。いわゆるページフォールトの例外である。

アドレス変換例外（ＡＴＲＥ）

Ａｄｄｒｅｓｓ　Ｔｒａｎｓｌａｔｉｏｎ　Ｅｘｃｅｐｔｉｏｎ（ＡＴＲＥ）

アドレス変換中のエラーによって発生する。具体的には、ＳＴＥ，ＰＴＥでｒｅｓｅｒｖｅｄのビットパターンを使用していた場合、ＵＡＴＢ，ＳＡＴＢ，ＳＴＥ，ＰＴＥによって未使用領域となっている部分を参照した場合、リング保護に違反したメモリアクセスを行なった場合などに発生する。ＥＩＴ発生の原因や詳細な情報は、ＡＴＲＥ発生時にスタックに積まれる情報によって区別される。

【０４１３】バスアクセス例外（ＢＡＥ）

Ｂｕｓ　Ａｃｃｅｓｓ　Ｅｘｃｅｐｔｉｏｎ（ＢＡＥ）

命令またはオペランドアクセスにおいて、一定時間以内にバスからの応答がなく、メモリアクセスができなかった場合に発生する。いわゆるバスエラーである。

奇数アドレスジャンプ例外（ＯＡＪＥ）

Ｏｄｄ　Ａｄｄｒｅｓｓ　Ｊｕｍｐ　Ｅｘｃｅｐｔｉｏｎ（ＯＡＪＥ）

分岐命令で、分岐先のアドレスが奇数であった場合に発生する。この例外は、ジャンプ先をオペランドとして直接指定する命令（ＪＭＰ，ＡＣＢ等）、スタックからリターンアドレスを得る命令（ＲＴＳ，ＥＸＩＴＤ，ＲＲＮＧ，ＲＥＩＴ）、およびＪＲＮＧ命令で発生し、ＥＩＴ処理の起動時には発生しない。ＥＩＴ処理起動時に新ＰＣが奇数であった場合には、システムエラー例外（ＳＥＥ）となる。［ＪＲＮＧ，ＥＩＴは詳細仕様調整中］

【０４１４】［命令、演算関係］

特権命令違反例外（ＰＩＶＥ）

Ｐｒｉｖｉｌｅｇｅｄ　Ｉｎｓｔｒｕｃｔｉｏｎ　Ｖｉｏｌａｔｉｏｎ　Ｅｘｃｅｐｔｉｏｎ（ＰＩＶＥ）

ｒｉｎｇ０以外から特権命令を実行しようとした場合に発生する。

〈〈Ｌ１〉〉機能例外（Ｌ１Ｅ）

Ｌ１　ｆｕｎｃｔｉｏｎ　Ｅｘｃｅｐｔｉｏｎ（Ｌ１Ｅ）

〈〈Ｌ１〉〉機能のインプリメントされていないプロセッサにおいて、〈〈Ｌ１〉〉の機能を実行しようとした場合に発生する。〈〈Ｌ１〉〉を実装しているプロセッサであれば、この例外は発生せず、このＥＩＴに対するベクトル番号はｒｅｓｅｒｖｅｄとなる。

予約命令例外（ＲＩＥ）

Ｒｅｓｅｒｖｅｄ　Ｉｎｓｔｒｕｃｔｉｏｎ　Ｅｘｃｅｐｔｉｏｎ（ＲＩＥ）

302

現在割り当てられていない命令やアドレッシングモードのビットパターンを実行しようとした場合に発生する。いわゆる未定義命令の例外である。「本発明装置」３２で６４ビットのサイズを指定した場合、Ｐビットを１にした場合、未実装の〈〈Ｌ２〉〉命令を実行しようとした場合、未定義、未実装のオプションを指定した場合も、これに含まれる。また、命令によって禁止されているアドレッシングモードを使用した場合　（ＪＭＰ命令におけるイミディエート指定など）や、インプリメントされていない段数の付加モードを使用した場合も、これに含まれる。

【０４１５】予約機能例外（ＲＦＥ）

Ｒｅｓｅｒｖｅｄ　Ｆｕｎｃｔｉｏｎ　Ｅｘｃｅｐｔｉｏｎ（ＲＦＥ）

命令やアドレッシングモードのビットパターン以外で、将来の拡張のために予約されている機能を利用しようとした場合に発生する。例えば、ＰＳＷに関しては、ＸＡやｒｅｓｅｒｖｅｄ（’－’）のビットに１を書き込んだ場合、ＳＭＲＮＧのフィールドに　ｒｅｓｅｒｖｅｄの値（ＳＭ，ＲＮＧ＝００１など）を書き込んだ場合、非特権命令のＬＤＰＳＢ，ＬＤＰＳＭによってＰＳＭやＰＳＢのｒｅｓｅｒｖｅｄ（’－’）のビットに１を書き込んだ場合などに予約機能例外（ＲＦＥ）が発生する。このほか、実装されていない制御レジスタをアクセスしようとした場合や、ＷＡＩＴ命令でｉｍａｓｋ≧１６を指定した場合にも、予約機能例外（ＲＦＥ）が発生する。なお、「本発明装置」では、命令ビットパターン（アドレッシングモードやサイズの指定を含む）のみでエラーと判定できるものを予約命令例外（ＲＩＥ）とし、アドレスやオペランド値によってエラーかどうかの状態が変化するものを予約機能例外（ＲＦＥ）としている。

【０４１６】コプロセッサ命令例外（ＣＩＥ）

Ｃｏ－ｐｒｏｃｅｓｓｏｒ　Ｉｎｓｔｒｕｃｔｉｏｎ　Ｅｘｃｅｐｔｉｏｎ（ＣＩＥ）

コプロセッサが接続されていないのに、コプロセッサ用に割り当てられた命令を実行しようとした場合に発生する。

コプロセッサコマンド例外（ＣＣＥ）

Ｃｏ－ｐｒｏｃｅｓｓｏｒ　Ｃｏｍｍａｎｄ　Ｅｘｃｅｐｔｉｏｎ（ＣＣＥ）

コプロセッサとのインタフェースでエラーがあった場合に発生する。

コプロセッサ実行例外（ＣＥＥ）

Ｃｏ－ｐｒｏｃｅｓｓｏｒ　Ｅｘｅｃｕｔｉｏｎ　Ｅｘｃｅｐｔｉｏｎ

コプロセッサの命令実行においてエラーがあった場合に発生する。

【０４１７】不正オペランド例外（ＩＯＥ）

Ｉｌｌｅｇａｌ　Ｏｐｅｒａｎｄ　Ｅｘｃｅｐｔｉｏｎ

303

（ＩＯＥ）
不合理なオペランドの指定を行なった場合に発生する。固定長ビットフィールド命令で３２（６４）ビット以上のｗｉｄｔｈを指定した場合などがこれに含まれる。奇数アドレスへのジャンプやゼロ除算も意味的には不正オペランド例外の一部と考えられるが、特別な意味を持つために異なる例外に分類されている。なお、「本発明装置」において命令のオペランドが不当な場合の対策としては、不当オペランド例外やゼロ除算例外とするケースのほかに、特にチェックを行なわないケース（ＣＨＫ命令の上限値と下限値の大小関係など）、適当な解釈をしてそのまま命令を実行するケース（シフト命令でｃｏｕｎｔが大きい場合など）、などがある。一方、命令を実行した結果が不当（オーバーフローなど）な場合には、ＥＩＴが起動されることはない。この場合は、Ｖ＿ｆｌａｇをセットして命令を終了するケース（ＡＤＤ，ＭＯＶなど多数）、特に何もしないケース（ＵＮＰＫｓｓでのオーバーフローなど）、などがある。
１０進不正オペランド例外（ＤＤＥ）
Ｄｅｃｉｍａｌ　Ｉｌｌｅｇａｌ　ＯＰｅｒａｎｄ　Ｅｘｃｅｐｔｉｏｎ（ＤＤＥ）
符号付き１０進演算命令において、０～９以外のデータをオペランドとして指定した場合に発生する。この例外も、意味的には不正オペランド例外（ＩＯＥ）の一部であるが、別の例外に分類されている。
【０４１８】予約スタックフォーマット例外（ＲＳＦＥ）
Ｒｅｓｅｒｖｅｄ　Ｓｔａｃｋ　Ｆｏｒｍａｔ　Ｅｘｃｅｐｔｉｏｎ　（ＲＳＦＥ）
ＲＥＩＴ命令によってＥＩＴから復帰する際に、ＥＩＴスタックフレームのフォーマットを示す番号（ＦＯＲＭＡＴ）が、ＲＥＩＴ命令で処理できないものであった場合に発生する。
リング遷移違反例外（ＲＴＶＥ）
Ｒｉｎｇ　Ｔｒａｎｓｉｔｉｏｎ　Ｖｉｏｌａｔｉｏｎ　Ｅｘｃｅｐｔｉｏｎ（ＲＴＶＥ）
ＪＲＮＧ命令で外側のリングに移ろうとした場合、ＲＲＮＧ命令で内側のリングに移ろうとした場合など、不当なリング遷移を行なおうとした場合に発生する。なお、ＪＲＮＧ命令で参照すべきＪＲＮＧＶＴＥを含むページが未使用領域となっていた場合には、リング遷移違反例外（ＲＴＶＥ）ではなくアドレス変換例外（ＡＴＲＥ）の未使用領域参照エラーが発生する。
ゼロ除算例外（ＺＤＥ）
Ｚｅｒｏ　Ｄｉｖｉｄｅ　Ｅｘｃｅｐｔｉｏｎ（ＺＤＥ）
０除算を行なった場合に発生する。
【０４１９】［デバッグ］
デバッグ例外（ＤＢＥ）
Ｄｅｂｕｇ　Ｅｘｃｅｐｔｉｏｎ（ＤＢＥ）

304

デバッグに関係して発生する。具体的には、命令のシングルステップ実行やブレークポイントを実現するための例外であるが、詳細な仕様は〈〈ＬＶ〉〉である。
［トラップ］
無条件トラップ命令（ＴＲＡＰＡ）
ＴＲＡＰＡ　Ｉｎｓｔｒｕｃｔｉｏｎ
ＴＲＡＰＡ命令により発生する。ＴＲＡＰＡのＥＩＴベクトルは、ＴＲＡＰＡのオペランドｖｅｃｔｏｒに対応して１６種類用意されている。
条件トラップ命令（ＴＲＡＰ）
Ｃｏｎｄｉｔｉｏｎａｌ　ＴＲＡＰ　Ｉｎｓｔｒｕｃｔｉｏｎ
ＴＲＡＰ命令により発生する。
【０４２０】［ＤＣＥ，ＤＩ］
遅延割り込み（ＤＩ）
Ｄｅｌａｙｅｄ　Ｉｎｔｅｒｒｕｐｔ（ＤＩ）
ＤＩレジスタ中のＤＩフィールドが、ＰＳＷ中のＩＭＡＳＫフィールドよりも小さい値になった場合に発生する。このＥＩＴは、コンテキストとは独立した非同期の事象を処理するために有効である。ＤＩ処理のＥＩＴベクトルは、各割り込み優先度毎に１５種類用意されている。このＥＩＴは、ＲＥＩＴ命令などの命令実行によって発生するという点では例外であるが、その時実行中のコンテキストと関係なく起動されるという意味では割り込みである。つまり、例外と割り込みの中間的なものである。（ＩＭＡＳＫフィールドを含むＰＳＷはコンテキスト依存であるが、ＩＭＡＳＫフィールドのみはコンテキスト独立として運用されるのが普通である。）詳しくは後の章を参照。
遅延コンテキスト例外（ＤＣＥ）
Ｄｅｌａｙｅｄ　Ｃｏｎｔｅｘｔ　Ｅｘｃｅｐｔｉｏｎ（ＤＣＥ）
ＣＳＷレジスタ（あるいはＤＣＥレジスタ）中のＤＣＥフィールドが、ＰＳＷ中のＳＭＲＮＧフィールドよりも小さい値になった場合に発生する。この例外は、コンテキストに依存した各種の非同期の事象（入出力の完了など）を処理するために有効である。詳しくは後の章を参照。
【０４２１】［その他］
リセット割り込み（ＲＩ）
Ｒｅｓｅｔ　Ｉｎｔｅｒｒｕｐｔ（ＲＩ）
外部からのリセット信号により発生する。
システムエラー例外（ＳＥＥ）
Ｓｙｓｔｅｍ　Ｅｒｒｏｒ　Ｅｘｃｅｐｔｉｏｎ（ＳＥＥ）
ＥＩＴ処理中に致命的なエラーが起きた場合に発生する。
［割り込み］
外部割り込み（ＥＩ）
Ｅｘｔｅｒｎａｌ　Ｉｎｔｅｒｒｕｐｔ（ＥＩ）

305

外部からのハードウエア信号により発生する。ＥＩＴベクトルは外部から指定する。一般に、外部割り込みのチェックは命令の切れ目で行なわれるのが普通である。しかし、「本発明装置」の場合は、実行時間の上限の決まらない高機能命令（任意長ビットフィールド命令、ストリング命令、ＱＳＣＨ命令）が存在する。これらの命令では、命令の実行途中であっても外部割り込みを受け付けるようになっている。

固定ベクトル外部割り込み（ＦＶＥＩ）

Ｆｉｘｅｄ　Ｖｅｃｔｏｒ　Ｅｘｔｅｒｎａｌ　Ｉｎｔｅｒｒｕｐｔ（ＦＶＥＩ）

外部からのハードウエア信号により発生する。ＥＩＴベクトルは優先度毎に一つずつ決まっている。いわゆるオートベクトルの割り込みである。

【０４２２】以上のＥＩＴのうち、予約例外、不正例外、違反例外の区別は次のような考え方によっている。予約ＸＸＸ例外機能拡張によって解消される可能性のあるもの将来の機能拡張によって、例外ではなくなる可能性がある。メーカー間のインプリメントの違いの出る場合がある。不正ＸＸＸ例外意味的に明らかにエラーであるもの予約例外とは異なり、将来の機能拡張があっても、永久の例外のままである。メーカー間のインプリメントの違いは出ないところである。ＸＸＸ違反例外リング保護の観点から実行を制限しているものその他ＯＳやシステム構成上の例外、複数の分類に当てはまる例外など

306

【０４２３】Ａ９－２．ＥＩＴのオペレーション

プロセッサがＥＩＴを検出すると、以下の手順にしたがってＥＩＴ処理を行なう。ただし、リセット割り込み（ＲＩ）、システムエラー例外（ＳＥＥ）については、これとは違った動作をする。また、以下の説明は「本発明装置」３２に限ったものであり、「本発明装置」６４では、パラメータ等が異なってくる可能性がある。

（Ｅ１）ベクトル番号の生成

プロセッサはそのＥＩＴに応じたベクトル番号をプロセッサ内部で生成する。ただし、外部割り込みＥＩの場合は、プロセッサの外部（周辺ＬＳＩなど）からＥＩＴベクトル番号を得る。

【０４２４】（Ｅ２）ＥＩＴＶＴＥの読み込み

「本発明装置」では、それぞれのＥＩＴに対する処理ハンドラの先頭アドレスと、ＥＩＴのベクトル番号との対応を示す表をＥＩＴベクトルテーブル（ＥＩＴＶＴ）と呼ぶ。また、そのテーブルの一つのエントリをＥＩＴＶＴＥと呼ぶ。「本発明装置」のＥＩＴＶＴＥは、ＥＩＴ処理の自由度と拡張性を考慮して８バイトとなっており、ＥＩＴ処理ハンドラの先頭アドレス（ＰＣ）だけではなく、ＰＳＷの一部のフィールドもセットすることができる。そのため、ＥＩＴＶＴＥはＰＣ＋ＰＳＷに準じた構成になっている。ＥＩＴＶＴＥのフォーマットは、図３５８のようになっている。

VS(Vector SM): EIT処理後のSM
　　ただし、VSがそのままEIT 処理後の SM となるのではない。詳細後述。
VX(Vector XA): EIT処理後のXA現在は 0 にreserved
　　（違反時は無視）
VAT(Vector AT): EIT 処理後のAT
VD(Vector DB): EIT処理後のDB
VIMASK(Vector IMASK): EIT 処理後のIMASK
　　ただし、VIMASKがそのままEIT 処理後のIMASK となるのではない。詳細後述。
VPC(Vector PC): EIT 処理後のPC
'=': 0 にreserved　（違反時は無視）
'-': 0 にreserved
　　（違反時はシステムエラー例外）

プロセッサは、（ＥＩ）で生成したＥＩＴベクトル番号に従い、

（EIT ベクトル番号）×８＋EITVTB

の物理アドレスにあるＥＩＴＶＴＥを読み込む。

【０４２５】（Ｅ３）ＰＳＷの更新

読み込んだＥＩＴＶＴＥをもとに、以下のようにＰＳＷを更新する。

［外部割り込み以外の場合］

min　(VS: 旧SM)　==>　新SM

スタックポインタの選択。

ＥＩＴ発生前にＳＰＩ以外のスタックポインタを使っていた場合は、ＶＳによってＥＩＴ処理ハンドラで使用するスタックポインタ（ＳＰ０またはＳＰＩ）が選択される。ＥＩＴ発生前に既にＳＰＩを使っていた場合には、ＶＳに関係なく、ＥＩＴ処理ハンドラでもＳＰＩをそのまま使う。このような仕様になっているのは、ＥＩＴが

307

ネストした場合を考慮したためである。

旧ＲＮＧ＝＝〉新ＰＲＮＧ

００＝＝〉新ＲＮＧ

ＥＩＴ処理ハンドラは、必ずリング０で実行される。なお、ＥＩＴＶＴＥには未使用のビットがあるので、将来的には、ＥＩＴによってリング０以外のリングに入るような指定をすることも可能である。

ＶＸ＝＝〉新ＸＡ

現在は０に固定されている。

ＶＡＴ＝＝〉新ＡＴ

ＥＩＴ処理ハンドラの実行中は、アドレス変換の有無を切り換えることができる。

ＶＤ＝＝〉新ＤＢ

ＥＩＴ処理ハンドラの実行中は、デバッグの環境を切り換えることができる。

ｍｉｎ（ＶＩＭＡＳＫ，旧ＩＭＡＳＫ）＝＝〉新ＩＭＡＳＫ

＊

min(VIMASK，発生した外部割り込みの優先度）　==>　新IMASK

この部分のみ外部割り込み以外の場合とは異なっている。この機能により、優先度の低い多重割り込みを禁止することができる。なお、割り込みマスクの機能により、

発生した外部割り込みの優先度＜旧ＩＭＡＳＫ

という関係が成立しているはずである。

【０４２７】（Ｅ４）プロセッサ情報のスタックへの退避

ＥＩＴ発生前の旧ＰＣ、旧ＰＳＷ、および発生したＥＩＴに関する各種の情報（ＥＩＴＮＦ－ＥＩＴベクトルやスタックフォーマットなどを含む）をスタックに退避する。この退避に使用されるスタックは、新ＳＭと新ＲＮＧ（＝００）により選択されるスタックである。この時に生成されるスタックフレームは、図３５９のようになる。このうち、ＥＩＴＩＮＦは、発生したＥＩＴにより生成されるスタックフレームのフォーマット（ＦＯＲＭＡＴ）、ＥＩＴのタイプ（ＴＹＰＥ）、ＥＩＴのベクトル番号（ＶＥＣＴＯＲ）などの情報を３２ビットに詰めたものである。追加情報の有無や内容は、ＥＩＴの種類によって異なる。ＲＥＩＴ命令では、ＥＩＴＩＮＦ中のＦＯＲＭＡＴを見ることによって追加情報の有無やそのフォーマットを知り、ＥＩＴ発生前のもとの命令列に戻るための情報を得る。なお、「本発明装置」６４になった場合に生成されるＥＩＴのスタックフレームは、旧ＰＣで１ロングワード、旧ＰＳＷとＥＩＴＩＮＦを合わせて１ロングワード、という構成になる予定である。ＥＩＴＩＮＦをＰＳＷに隣接した場所に置いたのは、「本発明装置」６４でのアラインメントの維持を考慮したためである。また、ＰＳＷをスタックトップに置いたのは、将来「本発明装置」６４で、３２ビットのコンテキストと６４ビットのコンテキストが混在した場合でも、スタック中に退避されたＰＳＷ中のＸＡビットが読み出せるようにするためである。

308

＊例外割り込みや内部割り込みによるＥＩＴの場合でも、ＥＩＴ処理でＩＭＡＳＫを操作することができる。この機能を使えば、ＥＩＴ処理に入るのと同時に外部割り込みを禁止することができる。したがって、ＥＩＴ処理と不可分に何らかの処理（例えばＥＩＴによって生成されたスタックフレームの転送など）を行ないたい場合に、この機能が有効である。

【０４２６】［外部割り込みの場合］

min （VS. 旧SM） ==> 新SM

旧RNG ==> 新PRNG

00 ==> 新RNG

VX ==> 新XA

VAT ==> 新AT

VD ==> 新DB

【０４２８】（Ｅ５）ＥＩＴ処理ハンドラの起動

ＶＰＣをＰＣに転送し、ＥＩＴ処理ハンドラを起動する。なお、命令プリフェッチにおいて発生したＥＩＴは、フェッチしようとした命令が必要になるまでＥＩＴ処理が遅らされる。これに対して、ＥＩＴ処理ハンドラの最後に置かれたＲＥＩＴ命令では、次のような処理を行なってもとの命令列に復帰する。

【０４２９】（Ｒ１）スタックからの読み込み

スタックより、旧ＰＳＷとＥＩＴＩＮＦを読み込む。読み込んだＰＳＷ中のＸＡビットが０であれば、ＥＩＴを発生したコンテキスト（タスクやプロセス）が３２ビットのコンテキストであったということがわかるので、続いてスタックより３２ビット幅で旧ＰＣを読み込む。なお、「本発明装置」、３２では、すべてのコンテキストが３２ビットのコンテキストである。さらに、ＥＩＴＩＮＦ中のＦＯＲＭＡＴにより追加情報の有無を判定し、追加情報があれば、スタックからそれを読み込む。追加情報には、ＥＸＰＣ，ＩＯＩＮＦ，ＥＲＡＤＤＲ，ＥＲＤＡＴＡ，ＳＰＩなどがあるが、その詳細な意味はインプリメント依存である。ＦＯＲＭＡＴがプロセッサのサポートしていない値（ＥＩＴで発生するはずのない値）であった場合は、予約スタックフォーマット例外（ＲＳＦＥ）となる。

【０４３０】（Ｒ２）ＰＳＷの復帰

スタックから読み込んだ旧ＰＳＷにより、ＰＳＷの全フィールド（ＳＭＲＮＧ，ＸＡ，ＡＴ，ＤＢ，ＩＭＡＳＫ，ＰＳＭ，　ＰＳＢ）をＥＩＴ発生前の値に復帰する。この時、旧ＰＳＷがｒｅｓｅｒｖｅｄの値を含んでいた場合には、予約機能例外（ＲＦＥ）を発生する。

（Ｒ３）ストアバッファの再実行（インプリメント依存）

309

ＦＯＲＭＡＴと追加情報の値によっては、ＲＥＩＴ命令の中で、前回ＥＩＴを発生したストアバッファによるライトサイクルの再実行を行なう場合がある。この時、ライトサイクルの実行に必要なアドレスとデータの情報としては、スタック上の追加情報の中にあったＥＲＡＤＤＲとＥＲＤＡＴＡが使用される。詳しくは、ＥＩＴのタイプの説明の項を参照。なお、ストアバッファのみを再実行する機能があるかどうかということは、プロセッサのインプリメント上の問題である。すなわち、ＲＥＩＴ命令のストアバッファ再実行機能を前提としてＥＩＴ処理が作られている場合には、当然ＲＥＩＴ命令でそれをサポートする必要があるし、ストアバッファのみの再実行機能が無くても矛盾のないＥＩＴ処理が実現できていれば、ＲＥＩＴ命令でそれをサポートする必要はない。プログラマから見た場合は、ＥＩＴ処理とＲＥＩＴ命令による処理がきちんと対応しており、ＲＥＩＴ命令によってＥＩＴ処理から戻った時に、ＥＩＴを発生した命令列が矛盾なく実行を続けられれば良いのである。この機能の有無と、プロセッサとしての機能の高低とは直接関係しない。

【０４３１】（Ｒ４）ＥＩＴ検出時に実行していた命令列への復帰

スタックから読み込んだ旧ＰＣをＰＣに復帰し、ＰＣの示す命令から実行を再開する。この時、ＥＩＴＩＮＦ中のＴＹＰＥフィールドによって、次に受け付けられるＥＩＴのタイプを変化させている。この機能は、多重ＥＩＴの処理を矛盾なく行なうために、また命令のシングルステップ動作を、エミュレーションによる実行を含めて正確に行なうために利用される。なお、ＥＩＴＩＮＦのＶＥＣＴＯＲフィールドは、ＲＥＩＴ命令では特に使用されない。にもかかわらずＶＥＣＴＯＲがＥＩＴＩＮＦに含まれているのは、ＥＩＴ処理ハンドラのプログラムに対して情報を提供するためである。

【０４３２】Ａ９－３．ＥＩＴのタイプ

「本発明装置」のＥＩＴを、ＥＩＴ処理ハンドラ終了後の実行再開時のＰＣの位置と、ＥＩＴ処理の優先度に着目して分類すると、以下のようになる。この分類は、ＥＩＴＩＮＦのＴＹＰＥフィールドの値にそのまま対応する。

［命令中断型ＥＩＴ（ＴＹＰＥ＝０，ＰＣ不定）］このＥＩＴが発生すると、直ちにそれが検出され、ＥＩＴ処理に入る。このＥＩＴが発生した場合、ＥＩＴを発生した命令系列に復帰することはできない。これに該当するのは、ＲＩ，ＳＥＥである。

［命令完了型ＥＩＴ（ＴＹＰＥ＝１～３，ＰＣ次命令）］このＥＩＴが発生すると、その時実行中の命令処理が完了した後でそれが検出され、ＥＩＴ処理に入る。一般には、このＥＩＴに対するＥＩＴ処理ハンドラの最後でＲＥＩＴ命令を実行することにより、ＥＩＴ発生時に実行していた命令の次の命令から、命令の実行を再開できる。なお、ＴＹＰＥ＝１～３の区別は、優先度などの関係によるものである。これに該当するのは、ＴＲＡＰ，ＴＲＡＰＡ，ＤＢＥ，ＤＩ，ＤＣＥなどである。

310

［命令再実行型ＥＩＴ（ＴＹＰＥ＝４，ＰＣ現命令）］このＥＩＴが発生すると、プロセッサやメモリの状態は、ＥＩＴ発生時に処理していた命令の実行開始前の時点に戻される。一般には、このＥＩＴに対するＥＩＴ処理ハンドラの最後でＲＥＩＴ命令を実行することにより、ＥＩＴ発生時に実行していた命令から、命令の実行を再開できる。これに該当するのは、ＰＯＥ，ＡＴＲＥ，ＢＡＥ，ＲＩＥ，ＲＦＥ，ＰＩＶＥ，ＩＯＥなどである。

【０４３３】命令完了型のＥＩＴは、以前実行していた命令に関するＥＩＴであり、命令再実行型のＥＩＴは、現在実行中の命令に関するＥＩＴである。したがって、複数のＥＩＴが同時に発生した場合には、一般に命令完了型のＥＩＴを先に処理する必要がある。また、命令中断型のＥＩＴは重要度の高いＥＩＴであり、これが検出された場合には、他のＥＩＴを処理することは無意味である。したがって、命令中断型のＥＩＴと他のＥＩＴが同時に発生した場合には、命令中断型のＥＩＴを先に処理する必要がある。結局、複数のＥＩＴが同時に発生した場合の優先度は

命令中断型＞命令完了型＞命令再実行型

となり、ＥＩＴＩＮＦのＴＹＰＥ＝０～４が、そのままＥＩＴの優先度を表わすことになる。ＥＩＴの種類とＴＹＰＥとの対応は、ＲＩ，ＴＲＡＰのように明確に決まっているものもあるが、ある程度はインプリメント依存の部分がある。したがって、ソフトウエアでＥＩＴの要因を分析する際には、できるだけＴＹＰＥのフィールドを参照したり、書き換えたりしない方がよい。

【０４３４】例えば、ページ不在例外（ＰＯＥ）の場合、これは命令再実行型のＥＩＴであり、ＴＹＰＥ＝４となるのが普通である。しかし、メモリの書き込みにストアバッファを用いるようなインプリメントのプロセッサにおいて、命令の最後の書き込みサイクル（ストアバッファ使用）でＰＯＥが発生した場合には、この命令全体を最初から再実行しなくても、最後の書き込みサイクルのみやり直せば処理上の矛盾は生じない。そこで、このようなケースでのＰＯＥを命令完了型のＥＩＴとし、エラーを発生した最後のライトサイクルの処理はＲＥＩＴ命令の中で行なうという場合がある。この場合は、ＰＯＥでもＴＹＰＥ＝１となる。また、ＥＩＴ処理でスタック積まれるＰＣは、ＰＯＥ発生命令ではなく次の命令になる。命令再実行方式に忠実にしたがう限り、命令実行中にエラーが発生した場合は、命令実行前の状態に戻してＴＹＰＥ＝４のＥＩＴを起動するというのが原則である。しかし、命令がもう少しで終了するというところでエラーが発生した場合には、一応命令を終わったことにしてＴＹＰＥ＝１のＥＩＴを起動し、残りの処理（ス

311

トアバッファのライトサイクル）はＲＥＩＴ命令に任せるというインプリメントも可能なのである。このようなインプリメント方法をとるのであれば、ＰＯＥのＴＹＰＥが１と４の２通りになる。この場合、ＴＹＰＥの違いによってＲＥＩＴ命令で必要となる処理が変わってくるので、ＲＥＩＴ命令はそれに対応できるようになっていなければならない。この方法では、命令最後のライトサイクルで発生したエラーによるＥＩＴに対して、命令全体を再実行するのではなく、最後のライトサイクルのみを再実行するという形になっている。つまり、一種の命令継続実行方式になっている。この場合、命令継続実行＊

312

＊のために退避される内部情報に相当するのが、ＥＩＴの追加情報としてスタック上に退避されるＥＲＡＤＤＲやＥＲＤＡＴＡである。なお、ストアバッファを用いた書き込みサイクルで発生するＥＩＴの場合、その書き込みを行なった命令の直後にＥＩＴ処理に入るとは限らない。

【０４３５】Ａ９－４．ＥＩＴのスタックフォーマット　ＥＩＴの検出にともなって、ＥＩＴ処理に必要な情　がスタックに退避される。そのスタックフォーマットは図３６０の通りである。

PSW　:EITを検出した時点のPSW
Format: スタックフォーマット番号(8bit)
Type　:EITタイプ(8bit)
Vector:EITベクトル番号（9bit）
PC　　:EITハンドラからの復帰後の、実行再開アドレス

この内、「その他の情報」は各ＥＩＴのスタックフォーマット番号に応じて異なり、ＥＩＴの要因を解析するための情報、ＥＩＴハンドラから復帰するための情報が含※まれている。スタックフォーマット番号に応じたスタックフォーマットを図３６１に示す。

PC　　：REIT 命令でEIT から戻った後に実行する命令の先頭アドレス
EXPC　：EITの検出時に実行していた命令のPC。
IOINF　：　入出力に関する情報
Error Addr:　EITを発生させたバスサイクルのアドレス
Error Data:　EITを発生させたバスサイクルのデータ（write のみ）
SPI　　：EIT検出時のSPI の値

【０４３６】フォーマット番号０：予約命令例外、予約機能例外、予約スタックフォーマット例外、リング遷移違反例外、〈〈Ｌ１〉〉命令例外、コプロセッサ命令例外、特権命令違反例外、不正オペランド例外、固定ベクトル外部割り込み、遅延割り込み例外、外部割り込み。
フォーマット番号１：バスアクセス例外、アドレス変換例外。
フォーマット番号２：デバッグ例外、奇数アドレスジャンプ例外、ゼロ除算例外、条件トラップ命令、トラップ命令。
フォーマット番号３：ＤＢＧ外部割込み、ＤＢＧトラップ命令、ＤＢＧデバッグ例外、ＤＢＧアクセス違反例外（つまりＤＢＧ　ＥＩＴ専用）
【０４３７】ＥＸＰＣは次のような目的で導入されたものである。
・エラー解析情報の提供
－ストアバッファの書き込みの際にＴＹＰＥ＝１のＥＩＴが発生したような場合、その書き込みを行った命令を指しているのが　ＥＸＰＣである。ＰＣは先に進んでしまっている。
－デバッグ例外では、ＰＣは次の命令を指し、ＥＸＰＣは前の命令を指している。したがって、例えば、ジャンプ命令実行時にデバッグ例外を起動するようにした場合、ＥＸＰＣによってジャンプ前のＰＣの値を、ＰＣによってジャンプ後のＰＣの値を知ることができる。
・多重ＥＩＴの処理
－ＴＹＰＥ＝１のＴＲＡＰＡなどのＥＩＴの場合は、その処理ハンドラの中でＥＸＰＣの情報が必要になることはない。しかしながら、ＴＹＰＥ＝１のＥＩＴ（ＴＲＡＰＡなど）とＴＹＰＥ＝２の　ＥＩＴ（デバッグ例外など）が同時に発生した場合、ＴＹＰＥ＝１のＥＩＴにおいて、ＴＹＰＥ＝２で使用するＥＸＰＣを退避しておかなければならない。ＴＲＡＰＡでもＥＸＰＣを退避するような仕様になっているのは、このためである。この場合、ＴＲＡＰＡの処理に対するＲＥＩＴ命令実行後のＥＸＰＣは、ＲＥＩＴ命令の先頭を指すのではなく、スタックからポップした旧ＥＸＰＣの値を復帰したものになっていなければならない。すなわち、ＲＥＩＴ命令の直後になっていたデバッグ例外を起動した直後にペンディングになっていたデバッグ例外を起動した場合、スタックに退避されるＥＸＰＣは、ＲＥＩＴ命令を指すのではなく、ＴＲＡＰＡ命令を指すものでなければならない。

313

（なお、この例は、ＴＲＡＰＡのＥＩＴＶＴＥでデバッグ例外をマスクすることを想定したものである。）

314

【０４３８】また、ＩＯＩＮＦの構成は図３６２のようになっている。

=　：'0'にreserved

WI　：REIT 命令におけるライトリトライの指示

メモリアクセス系のEIT

(TYPE=1) で意味を持つ。

WI=0　　ライトリトライ要

WI=1　　ライトリトライ不要

MEL：　アドレス変換例外の発生位置

0000　　エラーなし

0001　　アクセス権に関するエラー

0010～1110　(reserved)

1111　　I/O領域に関するアクセスエラー

MEC：メモリアクセス関係のエラーのエラーコード

0000　　エラーなし

0001　　未使用領域参照エラー

0010　　(reserved)

0011　　(reserved)

0100　　readに関するリング保護違反エラー

0101　　write に関するリング保護違反エラー

0110　　execute に関するリング保護違反エラー

0111　　(reserved)

1000　　read時のバスアクセス不能

1001　　write 時のバスアクセス不能

1010　　(reserved)

1011　　(reserved)

1100　　(reserved)

1101　　I/O 領域に対するメモリ間接アドレッシング

315　　316

1110　I/O 領域に対するexecute

1111　read時にI/O 領域とI/O 以外の領域をまたぐ
write 時にI/O 領域とI/O 以外の領域をまたぐ

RW　：　バスサイクル種別

RW=0　write

RW=1　read

BL　：　バスロック状態

BL=0　バスロック中でない

BL=1　バスロック中

PA　：　空間の指定

PA=0　(reserved)…論理空間（アドレス変換あり）

PA=1　物理空間（アドレス変換なし）

AT　：　EITが発生したバスサイクルのアクセスタイプ

AT=000　データ

AT=001　プログラム

AT=010　割込みベクトルフェッチ

AT=011～111　(reserved)

SIZ：　ライトリトライを行う際のデータサイズ

0000　(reserved)

0001　1 バイト

0010　2 バイト

0011　3 バイト

0100　4 バイト

0101～1111　(reserved)

【０４３９】Ａ９－５．「本発明装置」のＥＩＴベクトル・テーブル

図３６３、図３６４に示す

リセット割込みとＤＢＧモードのＥＩＴ（Ｎｏ．０～５）に関するＥＩＴテーブルのエントリはＳＰＩ値とＰＣ値で構成される。その他のＥＩＴに関するＥＩＴテーブルのエントリはＰＳＷ値とＰＣ値で構成される。ＥＩＴＶＢのリセット時の初期値は’ＦＦＦＦＦ０００’であるため、リセット割込みでは物理番地の’ＦＦＦＦＦ０００’からエントリ（ＳＰＩ，ＰＣ）がフェッチされる。

【０４４０】Ａ９－６．ＥＩＴ処理中のエラー

ＥＩＴ処理中（E-ＩＴ発生から状態の退避を経て新ＰＳＷの設定まで）に、別のＥＩＴが発生するような重大なエラーが起きた場合には、システムエラー例外（ＳＥＥ）となる。システムエラー例外（ＳＥＥ）となる可能性があるのは、ＥＩＴＶＴＥの読み込みに伴うバスアクセス例外、旧ＰＣ　、旧ＰＳＷの退避に伴うスタックのページ不在例外、アドレス変換例外などである。また、ＥＩＴＶＴＥのＶＰＣを含むワードのＬＳＢが’１’であった場合にも、システムエラー例外（ＳＥＥ）となる。システムエラー例外（ＳＥＥ）の発生は、ＳＰＩ，ＳＰ０のどちらかのスタックを使用するかということには関係しない、ＳＰ０のスタツクでページ不在例外が起った場合にも、ＳＰＩのスタックに切り換えてあるいはページ不在例外のＥＩＴＶＴＥで指定されるスタックに切り換えて）、ＥＩＴ処理中を継続するという仕様になっていない。一方、ＪＲＮＧによるリング遷移はＥＩＴでないので、ＪＲＮＧの処理中にページ不在例外が起った場合には、ページ不在例外のＥＩＴＶＴＥで指定されるスタックを使って、ページ不在例外のＥＩＴ処理中を行うことになる。この点で、ＥＩＴ処理中に含まれるＴＲＡＰＡとＥＩＴ処理中に含まれないＪＲＮＧでは、システムエラーとなるまでのステップが一段異なっているので、注意が必要である。（図３６５参照）

いずれにしても、ＯＳを作る場合には、ＳＰＩにより指定されるスタック領域はメモリ常駐とし、ＳＰ０により指定されるスタック領域も、特殊な使い方をする場合を除けばメモリ常駐となるようにプログラミングする必要がある。

【０４４１】Ａ９－７．多重ＥＩＴ

ＴＹＰ＝０のＥＩＴを除けば、ＥＩＴの検出とそれに対

317

する処理は、各命令の切れ目で行われる。したがって、場合によっては、命令の切れ目において複数のＥＩＴが同時に検出される可能性がある。これを多重ＥＩＴと呼ぶ。ここでは、多重ＥＩＴの処理順序について説明する。例えば、ＴＹＰ＝０のＴＲＡＰＡとＴＹＰ＝３の外字割り込み（ＥＩ）が同時に発生した場合、まずＴＲＡＰＡに対するＥＩＴ処理中が行なわれ、引続きＥＩに対するＥＩＴ処理中が行なわれる。その結果、ＰＣ，ＰＳＷとスタックの状態は図３６６のようになる。したがって、この例では、ＥＩＴ処理中終了後にまずＥＩの処理ハンドラが実行される。ＥＩの処理ハンドラが終了した後は、その最後に置かれたＲＥＩＴ命令により、一段浅いレベルにあるＴＲＡＰＡの処理ハンドラに移る。つまり、優先度の高いＴＲＡＰＡ処理ハンドラの方が後回しになるわけである。ただし、上の例ではＴＲＡＰＡのＥＩＴ処理中が先に行われるため、そこでＰＳＷを変更してＥＩをマスクすることができる。つまり、ＴＲＡＰＡのＥＩＴＶＴＥで

ＶＩＭＡＳＫ　＜　ＥＩの優先度

となるような指定を行っておけば、ＴＲＡＰＡのＥＩＴ処理中でＩＭＡＳＫが変更され、ＥＩに対するＥＩＴ処理中は行なわれなくなる。この場合は、ＴＲＡＰＡの処理ハンドラが実行される。そして、ハンドラの最後のＲＥＩＴ命令でＩＭＡＳＫが元の値に戻った時に、マスクされていたＥＩが起動されることになる。このように、優先度の高い（ＴＹＰＥの小さい）ＥＩＴ処理中におけるＰＳＷの更新によってマスクされるＥＩＴには、ＤＢＥ，ＥＩ，ＤＩ，ＤＣＥのＥＩＴ、つまりＴＹＰ＝２～３のＥＩＴがある。逆に言えば、マスク可能なＥＩＴ（処理要求を保持することが可能なＥＩＴ）が、優先度の低いＴＹＰ＝２～３になっているわけである。これに対して、ＴＲＡＰＡの場合は、ＥＩＴ処理中要求を保持するレジスタやハードウエアは何も用意されていない。ＰＣも次の命令に進んでいるので、ＴＲＡＰＡ命令を再実行することもできない。したがって、ＴＲＡＰＡ命令実行直後にＥＩＴ処理中を行わないと、ＥＩＴ処理中要求が失われてしまう。ＴＲＡＰＡをＴＹＰ＝１の高い優先度にしているのは、このためである。また、ＴＹＰ＝４のＥＩＴは命令再実行のＥＩＴであるため、他のＥＩＴに対する処理が終わった後でもう一度同じ命令を実行すれば、再び同じＥＩＴが発生する。命令実行型（ＴＹＰ＝４）のＥＩＴが最も低い優先度になつているのは、このためである。したがって、多重ＥＩＴの場合には、ＴＹＰ＝４のＥＩＴの処理は行う必要がない。ＴＹＰ＝４のＥＩＴ起動要求は、同時に発生しＴＹＰ＝１～３のＥＩＴの検出に伴ってキャンセルされる。ＲＥＩＴ命令実行直後に受け付けられるＥＩＴとは異なっている。ＲＥＩＴ命令では、スタック中からポップされたＥＩＴＩＮＦの中のＴＹＰＥによって、ＲＥＩＴ命令終了直後に受け付けるＥＩＴを調整している。ＲＥＩＴ命令実行後

318

に受け付けられるＥＩＴのＴＹＰＥは図３６７の通りである。このうち、ＴＹＰＥ＝２はデバッグ例外（ＤＢＥ）である。つまり、デバッグ例外に対するＥＩＴ処理中ハンドラのＲＥＩＴ命令実行直後には、デバッグ例外を受け付けないということを意味している。

【０４４２】ＲＥＩＴ命令実行直後かどうかによって、ＴＹＰＥ＝２のデバッグ例外の扱いが異なっているのは、１命令毎のシングルステップ実行を行うためである。この場合、デバッグ例外に対するＲＥＩＴ命令の直後で再びデバッグ例外を起こしていたのでは、被デバッグプログラムの実行が全く進まずに、デバッグ例外のみが続けて発生するという状況になってしまう。したがって、上記メカニズムにより、ＲＥＩＴ命令の直後ではデバッグ例外を発生せず、１命令実行してからデバッグ例外を発生するようにしているのである。一般に、シングルステップ実行を行う場合には、次に命令を実行するか、デバッグ例外を起動するかの二つの内部状態を持つ必要がある。「本発明装置」では、この二つの状態を、ＲＥＩＴ命令実行直後かどうかといった内部状態と、ＥＩＴのＴＹＰＥとの組み合せによって表現していると考えられる。なお、この考え方によるシングルステップ実行は、デバッグ例外と同時にほかのＥＩＴが発生した場合にも適用できる。予約命令例外（ＲＩＥ）のＥＩＴ処理中ハンドラで命令エミュレーションを行う場合は、他のＥＩＴ（ページ不在など）に対する処理ハンドラとは異なり、ＲＩＥの処理ハンドラの前後でデバッグ例外を起動しなければならない。例えば、シングルステップ実行後に通常命令→デバッグ例外→ページ不在例外であれば次は通常命令を実行する必要があるが、通常命令→デバッグ例外→予約命令例外（エミュレーション）であれば次はデバッグ例外の起動となる。これは、ページ不在例外がデバッガやデバッグ対象プログラムにとって全く見えないものであるのに対して、エミュレーション例外は、デバッガ対象プログラムにとって「一命令の実行」として見えるはずのものだからである。「本発明装置」の場合は、予約命令例外の　ＥＩＴ処理中ハンドラの中でＥＩＴＩＮＦのＴＹＰＥを調整することにより、以上のようなことが可能である。

【０４４３】Ａ９－８．「本発明装置」のＤＩ

Ａ９－８－１．ＤＩのオペレーション

「本発明装置」のＤＩ（Ｄｅｌａｙｅｄ　Ｉｎｔｅｒｒｕｐｔ）は、ＤＩレジスタ中のＤＩフィールドがＰＳＷ中のＩＭＡＳＫフィールドよりも小さい値になった場合に発生するＥＩＴである。この機能は、コンテキストとは独立した非同期の事象をペンディングとして処理要求だけを登録したり、処理順序をシリアライズしたりする時など有効なものである。ＤＩ処理のＥＩＴベクトルは、各割り込み優先度毎に１５種類用意されている。ＩＭＡＳＫの値と、そのフラッグ変化のその時に許可される外部割り込みとの関係は、図３６８のようになってい

319

る。ＤＩを起動するかどうかのチェックをする必要があるのはＩＭＡＳＫが大きくなった時、またはＤＩが小さ＊くなった時である。したがってこれに該当するのは、

```
LDC src,@psw    ;pswは制御空間中のPSW のアドレス
LDC src,@imask  ;imaskは制御空間中のimask のアドレス
LDC src,@DI     ;DI は制御空間中のDIのアドレス
REIT
WAIT
```

の各指令である。このうち、ＬＤＣ　ｓｒｃ，@ｄｉ以外の場合には、これらの命令を実行する前のＤＩフィールドの値が、起動されるＤＩのレベル（優先度）となる。ＤＩのレベルは、ＤＩとして起動されるＥＩＴのベクトル番号に影響する。また、ＬＤＣ　ｓｒｃ，@ｄｉによってＤＩが起動された場合には、ＬＤＣ実行前のＤＩフィールドの値ではなく、ＬＤＣによって新しくセットされたＤＩフィールドの値（つまりｓｒｃ）が起動されるＤＩのレベルとなる。なお、ＥＩＴ起動時（外部割り込み、例外、ＴＲＡＰすべて含む）にもＩＭＡＳＫの変化するところがあるが、この場合にはＩＭＡＳＫの値が大きくなることはないため、ＤＩは起動されない。ＤＩが起動された場合、ＤＩフィールドは１１１１（要求なし）にリセットされる。また、ＩＭＡＳＫフィールドは、受理されたＤＩのレベルを優先度とする外部割り込みが発生したのと同じ変化をする。つまり、ｍｉｎ（ＶＩＭＡＳＫ，受理されたＤＩのレベル）＝＝〉新ＩＭＡＳＫとなる。

※【０４４４】Ａ９－８－２．ＤＩの使用例

［例；本発明装置の遅延ディスパッチ（ｄｅｌａｙｅｄ　ｄｉｓｐａｔｃｈ）］本発明装置では、外部割り込み処理ハンドラの中から発行したシステムコールによってレディキューの状態が変わった場合に、それに伴うディスパッチング（レジスタの入れ替えなど）が割り込み処理ハンドラから戻るまで遅らされる。これは、多重割り込みに伴う矛盾を避けるためである。これをＤＩの機能によって実現する。

前提条件

・システムコールはＴＲＡＰＡのＥＩＴＶＩＥでは、ＶＩＭＡＳＫでは、ＶＩＭＡＳＫ＝１４を指定しておく。これは、ＤＩ機能によって、システムコール処理の最後のディスパッチングを行うためである。

・ディスパッチングの処理を行う部分は、ＤＩ１４によって起動される。

【０４４５】

※【数１８】

・'｜'は実行中の状態を表し・'：'は実行を中断した状態を表す。

【０４４６】一般のシステムコール処理

図３６９に示す。

外字割り込み処理ハンドラからのシステムコール

図３７０に示す。ＤＩの機能を使えば、本発明装置の遅延ディスパッチの処理をすなおに実現することができる。また、多重割り込みやシステムコールのネストが起こった場合にも容易に対処できる。

【０４４７】Ａ９－９．本発明装置のＤＣＥ

Ａ９－９－１．ＤＣＥのオペレーション

本発明のＤＣＥ（Ｄｅｌａｙｅｄ　Ｃｏｎｔｅｘｔ　Ｅｘｃｅｐｔｉｏｎ）は、ＤＣＥレジスタ（またはＣＳＷレジスタ）中のＤＣＥフイールドよりも小さい値になった場合に発生するＥＩＴである。この機能は、コンテキストに関連した非同期の事象（入出力の完了など）の処理をペンディングとして処理要求だけを登録したり、処理順序をシリアライズしたりする時などに有効なものである。ＤＣＥレジスタ（またはＣＳＷレジスタ）中のＤＣＥフィールドは、ＤＣＥ要求を入れるためのフィールドは、ＤＣＥ要求を入れるためのフィールドである。ＤＣＥレジスタ（またはＣＳＷレジスタ）はコンテキスト毎に固有のレジストであるため、コンテキスト毎に別々のＤＣＥ要求を与えることが可能である。ＤＣＥは各コンテキストに付随したものであるから、コンテキストとは独立した外部割り込みの処理中（ＳＭ＝０の場合）には、ＤＣＥは起動されない。また、他のコンテキストＡでより高い優先度のＤＣＥ要求が出ていても、そのコンテキストＡにディスパッチされない限り、コンテキストＡのＤＣＥは起動されない。別のコンテキストＢで出ているＤＣＥ要求がそれより低い優先度あつたとしても、コンテキストＢに先にディスパッチされれば、コンテキストＢのＤＣＥが先に起動される。ＤＣＥフィールドの値と、その時に起動されるＤＣＥとの関係は図３７１のようになっている。いずれの場合にも、ＳＭＲＮＧ＞ＤＣＥとなつた時にＤＣＥが起動される。（ｒｅｓｅｒｖｅｄ）の指定をした場合、実際にはＤＣＥ＝０００と同じ動作をする。ただし、将来の拡張のため、この機能を利用したプログラミングを行なってはいけない。

【０４４８】ＤＣＥの起動される可能性があるのは、ＳＭＲＮＧが大きくなつた時、またはＤＣＥフィールドの値が小さくなった時である。したがって、この条件に該当する以下の命令において、ＤＣＥを起動するかどうかのチエックをする必要がある。

```
321                                                    322
LDC  src, @psw      ; psw は制御空間中のPSW のアドレス
LDC  src, @smrng    ; smrng は制御空間中のSMRNG のアドレス
LDC  src, @dce      ; dce は制御空間中のDCE のアドレス
LDC  src, @csw      ; csw は制御空間中のCSW のアドレス
                    ; ただし、CSW は実装されない場合がある。
RBIT
RRNG
```

なお、ＥＩＴ起動時（外部割り込み、例外、ＴＲＡＰすべて含む）やＪＲＮＧ実行時にもＳＭＲＮＧの変化することがあるが、ＥＩＴやＪＲＮＧの場合はＳＭＲＮＧの値が大きくなることはないため、ＤＣＥが起動されることはない。ＤＣＥ自体は、一つのＥＩＴ処理として起動される。ＤＣＥのＥＩＴが起動された場合、ＤＣＥフィールドは１１１（要求なし）にリセットされる。ＳＭＲＮＧフィールドは、一般のＥＩＴ処理と同じように、ＤＣＥのベクトル番号に割り当てられたＥＩＴＶＴＥにしたがって変化をする。ＤＣＥはコンテキスト毎の処理であるため、起動されたＥＩＴ処理ハンドラでは、ＳＰＩではなくＳＰ０を使用するのが普通である。ＥＩＴＶＴＥの設定によっては、ＤＣＥ処理でＳＭ＝０（ＳＰＩ使用）に入ることも可能であるが、これは運用上の問題として対処し、ハードウエアでは特にチエックは行なわない。ＲＥＩＴ命令やＲＲＮＧ命令によつてＤＣＥが起動された場合、実際にＤＣＥを起動する処理はＲＥＩＴやＲＲＮＧと同時に行なわれてもよいが、動作仕様上は、一旦ＲＥＩＴやＲＲＮＧの実行が終わってからＥＩＴを起動するという形になる。たとえば、ＤＣＥ＝１１０の＊時にＲＲＮＧでｒｉｎｇ１からｒｉｎｇ３に戻ると、そこでＤＣＥが起動されてｒｉｎｇ０に入る。この時、ＰＲＮＧはｒｉｎｇ１ではなくｒｉｎｇ３となっていなければならない。ＤＣＥとＤＩや外部割り込みを比較すると、図３７２のようになる。入出力の完了通知などの場合には、外部割り込み処理ルーチンの中で、該当するコンテキストＤＣＥを起動するという流れになることもある。ＤＣＥをソフトウエアによってシミュレーションすることは不可能ではないが、一般には、スタック上に退避されたＰＳＷやＰＣの変更をしなければならないため、かなり面倒である。割り込んだプログラムが、割り込まれたプログラムのスタックフオーマツトをすべて知っていなければならないからである。

【０４４９】Ａ９－９－２．ＤＣＥのネスト

ＤＣＥは多重ネストができるとより効果の大きいものになる。したがって、ＤＣＥ要求が複数発生した場合に、どのような処理をするかが問題となる。本案装置では、ネストの処理（要求のキユーイング）はソフトウエアで行なう方針である。

＜＜複数のＤＣＥ要求のキューイング処理例＞＞

```
〔DCE 要求設定時〕
 if (DCB=111 ) then
   新しいDCE 要求==> DCE フィールド／*DCE要求これだけ* ／
 else
   新しく発生したDCB 要求を、リング順に構成されたDCB 要求キューに入れる
endif
〔DCE 処理時〕
   ／*DCE起動の際、ハードウエアにより111==>DCB となる* ／
   if (DCE 要求キューが空でない) then
        DCE 要求キューの次のエントリをDCB フィールドにセット
   endif
```

【０４５０】Ａ９－９－３．ＤＣＥの使用例

〔例；入出力管理プログラムの起動〕外部割り込みによって入出力完了が通知され、それによつて、プロセスＡの入出力管理部（ｒｉｎｇ１）がプロセスＡに対して非※同期に起動されるものとする（図３７３参照）。

【０４５１】

【数１９】

'・' は実行中の状態を表し ':' は実行を中断した状態を表す。

【０４５２】①の開始アドレスはプロセス（コンテキスト）毎に指定されるべきものだが、実際にはＤＣＥのＥＩＴ処理ベクトルがプロセス間で共通であるため、ＯＳでプロセス毎のＤＣＥ要求テーブルを解析し、そこへジヤンプする必要がある。この図の場合は、外部割り込みが発生した時にたまたまプロセスＡが実行中だつたわけ

323

である。他のプロセスの実行中に入出力の外部割り込みが発生した場合、ｒｉｎｇ１の入出力管理部の起動は、プロセスＡへのデイスパッチが行なわれるまで遅延させられる。

324

＊【０４５３】付録１０．本発明装置の命令ビットパターン

〔表記法に関する注意〕命令ビットパターンの表記は、次のように行なう。

'-'　0 にreserved（違反時例外発生）

'+'　1 にreserved（違反時例外発生）

このビットが0(1)であれば処理が正常に行なわれ、このビットが1(0)であれば予約命令例外(RIE) を発生する。

'='　0 にreserved（違反時も無視)...Ver0.87 の '*'

'#'　1 にreserved（違反時も無視）

ユーザへのマニュアルには、将来の拡張のためこのビットを0(1)にしておくように明記する。実際には、このビットが0(1)であっても1(0)であっても動作は同じである。

「違反時も無視」というのは、アーキテクチャ上あまり好ましいことではないが、命令ビットパターンの割り当て、将来の拡張性、命令の高速実行の上からやむを得ない場合がある。

'~'　0 にreserved（違反時動作を保証しない）

'!'　1 にreserved（違反時動作を保証しない）ユーザへのマニュアルには、将来の拡張のためこのビットを0(1)にしておくように明記する。実際には、このビットが0(1)の場合は正常な動作を行なうが、1(0)の場合の動作はインプリメントに依存して異なっている。

「違反時動作を保証しない」というのは、アーキテクチャ上あまり好ましいことではないが、インプリメントや命令ビットパターンの割り当て、命令の高速実行の上からやむを得ない場合がある。例えば、LDATE.MULXの第一ハーフワードの'!R'がこれに該当する。

【０４５４】Ａ１０－１．命令のフォーマット別ビット割り当て

〔ビット割り当てに関する注意〕

・本発明装置では使用できるアドレッシングモードが命令毎にかなり異なっており、そのチェックをする必要がある。チェックを容易にするため許されるアドレッシングモードが区別しやすいようにビットパターンの割り当てを行なう。特定のアドレッシングモードを禁止しているオペランドの場合は、原則としてそれがそのオペランドを含むハーフワードだけで分かるようになっている。

・Ｐビットは、原則としてオペランド毎（レジスタ直接指定とイミディエート指定を除く）、および暗黙のスタック参照について一つずつ独立に入れる。命令パターン中では、'Ｐ'または'Ｑ'で表す。ただし、一般形の命令でカバーできる場合は、同じ命令の短縮形ではＰビットが入らないこともある。（ＰＵＳＨ，ＰＯＰ，ＰＵＳＨＡのみは、スタック参照に関するＰビットがない。）

・ビットパターンのうちＬＶｒｅｓｅｒｖｅｄで示されているものは、各メーカーで自由に使用してよい命令ビットパターンである。この命令ビットパターンは、例えばＩＣＥとのインターフェースを行なうための、ユーザに解放しない命令として利用することができる。以下図３７４～図３８５に示す。

【０４５５】Ａ１０－２．予約命令例外の検出について

上記のビットパターン中で｛ＲＩＥ｝で示されたパターンは、将来の拡張のために予約された（ｒｅｓｅｒｖｅｄ）のビットパターンである。この命令ビットパターンを実行すると、予約命令例外（ＲＩＥ）は発生する。このほか、インプリメントされていないオプションやサイズ（未実装の〈〈Ｌ２〉〉を含む）を指定した場合、未定義のオプションを指定した場合、命令ビットパターンの'－'の部分を'１'にした場合、命令ビットパターンの'＋'の部分を'０'にした場合、命令中の'Ｐ'，'Ｑ'のビットを'１'にした場合、ｒｅｓｅｒｖｅｄの条件（ｃｃｃｃ）や終了条件（ｅｅｅｅ）を指定した場合にも、すべて予約命令例外（ＲＩＥ）となる。現在、ＬＤＡＴＥ，ＭＵＬＸなどの例外を除けば、原則として第一～第四バイトについてはすべての命令パターンをチェックし、パターンが違っている場合には、

325

ＲＩＥにしている。第五、第六バイトについてはチェックを行わず、パターンが違っていてもエラーとはしていない。上記のビットパターンのうちで、特に〔ＲＩＥ－Ｘ〕で示したものは、第一ＨＷが汎用アドレッシングモードを含み、第二ＨＷまで読まないとＲＩＥであることがわからないビットパターンである。この場合は、第一ＨＷのＥａの拡張部の後に第二ＨＷが置かれる。現在使用していないビットパターンのうち、将来の機能拡張にともなって実装の予定のあるパターンや、他のメーカーのチップと動作の異なりそうなパターンについては、特に積極的に検出を行なうべきである。これは、その命令パターンを実行した場合の間違いを防ぐためである。このような方針に基づいた場合、予約命令の例外（ＲＩＥ）のチェックの優先度は次のようになる。

【０４５６】↑高優先度
（既に意味が決まっているもの）
未定義の〈〈Ｌ２〉〉機能の指定
６４ビットサイズの指定（ＲＲ，ＭＭ，ＷＷ，ＳＳ＝１１）
（命令の拡張に利用される可能性の高いもの）
〔ＲＩＥ〕となっている命令パターンの指定
ＢＶＰＡＴ～ＢＶＳＣＨの’＋Ｘ’の’＋’
ＰＳＴＬＢ～ＥＸＩＴＤ：Ｇのグループの第二ＨＷの’－’Ｐビット指定
（命令の拡張にはまず利用されないもの）
ＬＤＡＴＥ～ＩＮＤＥＸのグループの第一ＨＷ’！Ｒ’の’！’
ＳＴＡＴＥ～ＱＩＮＳのグループの第二ＨＷ’＋Ｗ’の’＋’
ＰＳＴＬＢ～ＥＸＩＴＤ：Ｇのグループの第一ＨＷ’＋Ｘ’の’＋’
ＡＣＢ：Ｒ，ＳＣＢ：Ｒの第二ＨＷの’－’
↓低優先度
現在チェックすべきビットパターンは前述のような仕様になっているが、今後、このような方針に基づいて予約＊

326

＊命令例外の検出に関する詳細仕様の調整を行ない、一部は変更の行われる可能性がある。なお、命令をどこまで読んだ時にＥＩＴを起動するかということは、特に規定しないものとする。第一ＨＷだけでＥＩＴを起動することが明らかである場合にも、第二ＨＷまで読んでもよい。また、オペコード部だけでＥＩＴを起動することが明らかである場合（予約命令例外など）に、Ｅａの拡張部まで処理しても構わない。

【０４５７】Ａ１０－３．オペランドフィールド名索引
図３８７、図３８８に示す。

【０４５８】Ａ１０－４．アドレッシングモードのビット割り当て
共通ビットパターン
・サイズ関係
01　16 bit
10　32 bit
11　64 bit

・アドレッシングモード
00　@reg+など
01　16 bit 相対間接モード
10　32 bit 相対間接モード
11　付加モード

・レジスタ指定
00　〔特殊〕
01　(SP)
10　absまたは0
11　PC

【０４５９】付加モード
ei<Rx>MS PXXD<d4>
'-'は 0にリザーブされるビット

<Rn>0* *XX***	Rn がインデクス
** −01* * −*****	インデクスなし
** −11* * −*****	RCがインデクス
	XX ≠00によるスケーリングは利用できない
******** ***0<d4>	4ビットディスプレースメント
******** ***1−01	16 ビットディスプレースメント
******** ***1−10	32 ビットディスプレースメント
******** ***1−11	64 ビットディスプレースメント

（ｄ４）のサイズ指定部分とＭＩＳＣモードのｄｉｓｐ：１６，ｄｉｓｐ：３２の指定部分が同じビット位置になっている。

【０４６０】基本モード

327　　328

```
P000 xxxx   MISC                P=0:Sh
     0000   {RIE }
     0001   {RIE }
     0010   {RIE }
     0011   {RIE } - @ads:64
     0100   @SP+(read:@sp+,write:illegal,rmw:illegal)
     0101   @SP+(write:illegal,read:@sp+,rmw:illegal)
     0110   {RIE }
     0111   {RIE }
     1000   {RIE }
     1001   @ads:16
     1010   @ads:32
     1011    絶対付加モード
     1100   [mm+(read:@sp+,write:illegal,rmw:illegal)
     1101   @(disp:16,Rn)
     1110   @Rn
     1111   @(disp:32,Rn)

0001 <Rn>   Rn                      Sh
1001 xxxx   {RIE }
P010 <Rn>   @(disp:16,Rn)           P=0:Sh
P011 <Rn>   @Rn                     P=0:Sh
P100 <Rn>   @(disp:32,Rn)
P101 <d4>   @(disp:4,FP)  <<L2>>
P110 <Rn>    レジスタ相対付加モード
P111 <d4>   @(disp:4,SP)  <<L2>>
```

〔Ea〕　　〔Sh〕

```
0000 00**          00 00**
0000 011*          00 011*
0000 1000          00 1000
0101 **** (<<L2>>未実装時のみ)
0111 **** (<<L2>>未実装時のみ)
1*** ****
```

・＊＊＊１　＊＊＊＊のパターンの時には拡張部が付かない。

・@ａｄｓ：６４の割り当てが変則的であるが、６４ビット拡張部に対しては、おそらく内部の回路でも特殊な扱いが必要であると思われる（例えば、本発明装置でも　@（ｄｉｓｐ：６４，Ｒｎ）は基本モードで実現できない）ので、それほど問題ではないと思われる。むしろ、付加モードとの関係が統一的になるようにした。

【０４６１】未定義のアドレッシングモードを指定した場合（ＥＡ中のＰビット＝１を含む）には、予約命令例外（ＲＩＥ）となる。具体的には、以下のパターンの場合にＲＩＥとなる。

30　付加モードでリザーブのパターンを指定した場合にも、予約命令例外（ＲＩＥ）となる。Ｍ＝１で〈Ｒｎ〉≠００００，０００１の場合、Ｄ＝１で〈ｄ４〉≠０００１，００１０以外の場合、Ｐ＝１の場合、ＸＸ＝１１の場合もこれに含まれる。付加モードのある段で、ＰＣに対して、×２、×４、×８以外のスケーリングを指定した場合には、その段の終了処理後の中間値として、インプリメントに依存した不定値が入る。ＥＩＴにはならない。〈〈Ｌ２〉〉未定義で、５段以上の付加モードを指定した場合にも、予約命令例外（ＲＩＥ）となる。〔詳

40　細調整中。予約機能例外ＲＦＥとなる可能性もある。〕命令によって使用できないアドレッシングモードの組み合わせを指定した場合（ＪＭＰ　＃ｉｍｍ＿ｄａｔａ，ＣＭＰ＃０，＃１など）にも、予約命令例外（ＲＩＥ）となる。〈〈Ｌ２〉〉未実装のために実行できない組み合わせを指定した場合も、これに含まれる。（これに当てはまるものは、レジスタ指定のビットフィールド命令である。）

【０４６２】Ａ１０－５．命令オプションのビット割り当て

50　いずれの場合にも、最初に記述した方（オプション値が

329　　330

０，００．．の方）がアセンブラでのデフォルトにな　　る。

cccc　Bcc.TRAP/cc での条件指定

eeee　ストリング命令、QSCH命令での終了条件指定

p,q..　P ビット指定(Q..は、P ビットの必要なオペランドが複数の場合)

b　　/F=0,/B=1(BSCH,BVSCH,BVMAP,BVCPY,SCMP,SMOV,QSCH)

r　　/F=0,/R=1(SSCH)

c　　/N=0,/S=1(CHK)－CHK,change index valueの 'c'

d　　/0=0,/1=1(BSCH,BVSCH) －dataの 'd'

m　　/NM=0,/MR=1(QSCH) －maskの 'm'

p　　/AS=0,/SS=1(PTLB,PSTLB,LDATE) －PTLB,specific space の 'p'

ttt　　/PT=000,/ST=001,/AT=110, {RIE } =010～101,111(PSTLB,LDATE,STATE)

xx　　/LS=00,/CS=01, {RIE } =10,11(LDCTX, STCTX)

【０４６３】Ａ１０－６．Ｂｃｃ命令、ＴＲＡＰ／ｃｃ命令の条件指定（ｃｃｃｃ）

ｃｃｃｃの値の割り当ては、図３８９のようになる。

【０４６４】Ａ１０－７．終了条件の指定（ｅｅｅｅ）

ｅｅｅｅの値の割り当ては、図３９０のようになる。００００～０１０１については、ｃｃｃｃ（Ｂｃｏｎｄ命令の条件指定）のビットパターンに意味を合わせてある。特に、ＬＴＵ，ＧＥＵは、ＳＵＢＸ命令などにおいて、オペランドを符号なしデータと考えた場合の比較結果がＸ＿ｆｌａｇに反映されるのに合わせたものである。なお、〈〈Ｌ２〉〉の終了条件のうち、二つの条件が．ｏｒ．で結ばれているものについては、そのうちのいずれの条件で終了したかを示すために、Ｍ＿ｆｌａｇを使用する。Ｍ＿ｆｌａｇがセットされるのは、原則として　Ｒ４との比較によって終了した時であり、具体的には図３９１のような場合である。Ｍ＿ｆｌａｇ＝１の条件を満たさなかった場合、および、これ以外の終了条件で終了した場合には、Ｍ＿ｆｌａｇ＝０となる。

〈〈Ｌ２〉〉の終了条件をインプリメントしない場合には常にＭ＿ｆｌａｇ＝０となるが、その場合でも、将来はＭ＿ｆｌａｇ≠０となる場合が生じるということを、マニュアルに明記しておくのが望ましい。

【０４６５】Ａ１０－８．ＢＶＭＡＰ命令の演算コードＲ５の下位４ビットに入れる演算コードである。これを図３９２に示す。

【０４６６】Ａ１０－９．アドレッシングモード対応

各命令のオペランドと、禁止されているアドレッシングモードモードとの対応を図３９３～図３９５に示す。○の組み合わせに対しては、そのアドレッシングモードモードが使用可能である。×の組み合わせに対しては、それを実行しようとした場合に予約命令例外（ＲＩＥ）が発生する。

【０４６７】付録１１．高機能命令の詳細仕様と終了時のレジスタ値

各命令の解説の項では、高機能命令の詳細や終了時のレジスタ値について明確に述べられていないので、ここでまとめて説明を行なう。

【０４６８】Ａ１１－１．高機能命令の仕様決定の方針

ＳＭＯＶ／Ｂ，ＳＣＭＰ／Ｂ，ＢＶＭＡＰ／Ｂ，ＢＶＣＰＹ／Ｂでは、@－ＳＰとの対応などからプリデクリメントの形で処理を行なうという考え方と、ＳＭＯＶ／Ｆ，ＳＳＣＨ／Ｒなどとの整合性からポストデクリメントの形で処理を行なうという考え方とがある。例えば、Ｈ’１００～Ｈ’１ｆｆの領域をＳＭＯＶ／Ｂ．Ｂによって転送する場合、ＳＭＯＶ／Ｂがプリデクリメントの仕様であればレジスタの初期値はＨ’２００となるし、ＳＭＯＶ／Ｂがポストデクリメントの仕様であればレジスタの初期値はＨ’１ｆｆとなる。

［ポストデクリメントのデメリット］ＳＭＯＶ／ＦとＳＭＯＶ／Ｂ，ＳＣＭＰ／ＦとＳＣＭＰ／Ｂとの対称性が悪くなる。例えば、Ｈ’０００００ｆｆまでの領域を占めるストリングに対してＳＭＯＶ／Ｂを実行する場合、ＳＭＯＶ／Ｂ．Ｂであればポインタの初期値としてＨ’０００００ｆｆを設定し、ＳＭＯＶ／Ｂ．Ｗであればポインタの初期値としてＨ’０００００ｆｃを設定する必要がある。

【０４６９】［プリデクリメントのデメリット］ＳＳＣＨ，ＢＳＣＨなどのサーチ系の命令との整合性が悪くなる。もし、ＳＳＣＨで、命令終了後のポインタ最終値が必ず終了条件成立のエレメント（サーチ結果のエレメント）を指すという原則を設けるとすると、／Ｆ，／Ｂ，／Ｒといった処理方向によってプリ更新／ポスト更新を変えることはできなくなる。したがって、／Ｂのみプリデクリメントとするわけにはいかない。（実際にはＳＳＣＨ／Ｂは存在しないが、ＢＳＣＨ／Ｂなどの仕様との関連がある）

ＴＲＯＮＣＨＩＰでは、［ポストデクリメントのデメリット］の方を重視し、ＳＭＯＶ／Ｂ，ＳＣＭＰ／Ｂではプリデクリメントの仕様にする。次に、ＳＭＯＶ，ＳＣＭＰ，ＳＳＣＨが終了条件によって終了した場合、ポイ

331

ンタの更新を行なってから命令を終了するか、ポインタの更新前に命令を終了するか、という問題がある。

【０４７０】［ポインタの更新前に命令を終了するデメリット］エレメントサイズによって命令を終了する場合には、ポインタの更新が行なわれ、ポインタが次のエレメント（／Ｆの場合はまだ処理の終わっていないエレメント）を指すようになってから命令を終了するので、その仕様とは合わなくなる。つまり、終了条件が成立するかどうかによってポインタを更新して良いかどうかの状況が変わるため、仕様がわかりにくくなる上、高速なインプリメントが難しくなる。また、ＳＳＣＨで、サーチが成功してから連続して次のサーチを行なう場合には、次のＳＳＣＨ実行前に別命令でポインタの更新が必要になる。ＳＭＯＶ，ＳＣＭＰでも同様である。

【０４７１】［ポインタの更新後に命令を終了するデメリット］命令実行後のポインタ値が終了条件（サーチ条件）成立のエレメントよりも進んでいるので、ＳＳＣＨ命令としてはすなおな仕様ではない。ＢＶＳＣＨ，ＢＳＣＨ命令の仕様とも合わない。ＴＲＯＮＣＨＩＰでは、［ポインタの更新前に命令を終了するデメリット］の方を重視し、終了条件成立の場合には、ポインタの更新後＊

332

＊に命令を終了するという仕様にする。したがって、ＳＭＯＶ／Ｆ，ＳＣＭＰ／Ｆ，ＳＳＣＨ／Ｆ，／Ｒ命令終了後のポインタは、終了条件の成立したエレメントの次のエレメントを指すことになる。また、ＳＭＯＶ／Ｂ，ＳＣＭＰ／Ｂ命令の場合はプリデクリメントでポインタを更新するため、命令終了後のポインタは終了条件の成立したエレメントを指すことになる。ＳＭＯＶ／Ｂ，ＳＳＣＭＰ／Ｂとの仕様を合わせるという意味で、ＢＶＭＡＰ／Ｂ，ＢＶＣＰＹ／Ｂの場合にも、演算の対象となるビットフィールドの最大オフセット＋１をＲ１，Ｒ４で指定する。

【０４７２】ただし、ＢＶＳＣＨ，ＢＳＣＨについては、命令終了後のビットオフセットが直接サーチ対象のビットを指している方が便利だと考えられるため、／Ｆ，／Ｂともそのような仕様にする。また、ＱＳＣＨについては、ポインタがプリ更新となっているため、ＳＳＣＨ，ＢＳＣＨとはポインタ更新のタイミングが異なっている。結局、ＢＳＣＨ／Ｆ（ＢＶＳＣＨ／Ｆ），ＳＳＣＨ／Ｆ，ＱＳＣＨ／Ｆのサーチのパターンをまとめると、次のようになる。

BSCH/F　現在ポインタの指しているものからサーチする。
サーチ終了後のポインタは、見付かったデータを指す。

SSCH/F　現在ポインタの指しているものからサーチする。
サーチ終了後のポインタは、見付かったデータの次を指す。

QSCH/F　現在ポインタの指している次のものからサーチする。
サーチ終了後のポインタは、見付かったデータを指す。

ストリング命令の場合、エレメント数Ｒ２は符号なしの数として扱われる。これは、Ｒ２を符号なしと考えることにより、Ｒ２＝０の指定によってエレメント数をＨ’１００００００００と解釈し、エレメント数による終了を行なわないようにできるためである。この機能は、言語Ｃのｓｔｒｃｍｐ関数の実現などに利用できる。また、インプリメント上も、Ｒ２を符号なしと考える方がエレメント数による終了の判定が楽になる。一方、ビットフィールド命令のｗｉｄｔｈは、以下のような理由により、固定長ビットフィールド命令、任意長ビットフィールド命令とも符号付きとして扱うことにする。

【０４７３】・命令実行時、ビットフィールド命令のｗｉｄｔｈはｏｆｆｓｅｔに加算される形になるが、ｏｆｆｓｅｔは符号付きである。ｗｉｄｔｈを符号なしとすると、符号付きと符なしの数を足すことになり、すっきりしない。ストリング命令のエレメントサイズの場合は、サイズを乗じた上でポインタに加算するのであるから、符号なしの方が自然である。

・命令実行の上で符号付きか符号なしかの違いが出てくるのは、任意長ビットフイールド命令でｗｉｄｔｈがＨ’８０００００００～Ｈ’ｆｆｆｆｆｆｆｆの場合である。ｗｉｄｔｈを符号付きとしておけば、ｗｉｄｔｈがこの値の時Ｖ＿ｆｌａｇをセットするだけで命令を終了するが、ｗｉｄｔｈを符号なしとした場合には、ｗｉｄｔｈがこの値の時もビットフィールド操作を行なうことになる。しかし、ｗｉｄｔｈがＨ’８０００００００～Ｈ’ｆｆｆｆｆｆｆｆだとすると、ｏｆｆｓｅｔ＋ｗｉｄｔｈの値を符号付きとして扱う場合には既にオーバーフローしているし、ｏｆｆｓｅｔ＋ｗｉｄｔｈを符号なし（あるいは３３ビット符号付き）として扱う場合にも、ｏｆｆｓｅｔの値によってはオーバーフローする。オーバーフローに関しては、「ｏｆｆｓｅｔ＋　ｗｉｄｔｈがオーバーフローした場合には動作を保証しない」となっているのであるから、同じようなインプリメントをするのであれば、ｗｉｄｔｈを符号なしとしても「動作を保証しない」ケースが増えるだけである。ｗｉｄｔｈを符号なしとして、しかもｗｉｄｔｈ）Ｈ’８０００００００の場合の動作を保証するのであれば、ハードウエアの負担を伴う。

・ストリング命令の場合は、終了条件によって命令を終了する場合があったため、エレメントサイズによる終了をしたくない時に０を設定するという使い方がある。０で無限大（Ｈ’１００００００００）を表現するには、どうしてもエレメントサイズを符号なしとして扱う必要

333

があった。これに対して、ＢＶＭＡＰ，ＢＶＣＰＹではｗｉｄｔｈ以外に命令終了の要素がないため、プログラミング上、ｗｉｄｔｈとして必ず意味のある値を設定することになる。その場合は、「レジスタ上の値は原則として符号付きの数と考える」という原則に合わせる方が自然である。

【０４７４】［ストリング命令、任意長ビットフィールド命令の基本原則のまとめ］

・サーチ系の命令では、ポインタ更新のタイミングはサーチ方向に依存しない。ＢＳＣＨ，ＢＶＳＣＨでは、／Ｆ，／Ｂともサーチ終了後のポインタは見付かったビットを指す。ＳＳＣＨでは、／Ｆ，／Ｒともサーチ終了後のポインタは見付かったエレメントの次のエレメントを指す。

・実際にデータを操作する命令では、／Ｆの場合にポストインクリメント、／Ｂの場合にプリデクリメントの形で処理を行なう。これに該当するのは、ＳＭＯＶ，ＳＣＭＰ，ＢＶＭＡＰ，ＢＶＣＰＹである。ＳＳＴＲ，ＢＶＰＡＴは／Ｆのみであるが、やはり同じ原則にあてはまる。

334

・ストリング命令では、エレメントサイズは符号なしとして扱われ、’０’の場合はＨ’１０００００００を表わす。一方、任意長ビットフィールド命令では、ｗｉｄｔｈは符号付きとして扱われ、ｗｉｄｔｈがＨ’００００００１～Ｈ’７ｆｆｆｆｆｆｆの場合にのみ実際のビットフィールド操作を行なう。

【０４７５】Ａ１１－２．ストリング命令の詳細仕様

ＳＭＯＶ

ＳＭＯＶのオペレーションをまとめると、次のようになる。ただし、最終的な結果が同じであれば、メモリアクセスの順番が下記のものと異なっていても構わない（他の高機能命令も同様）。また、ｓｒｃとｄｅｓｔがオーバーラップしていた時に、正しくない方のオプションを使用した場合（ｓｒｃ　＜ｄｅｓｔで／Ｆを使用した場合、およびｓｒｃ　＞ｄｅｓｔで／Ｂを使用した場合）の動作も、下記の通りでなくてもよい。

【０４７６】［ＳＭＯＶ／Ｆのオペレーション］

```
0 ==>  V__flag
repeat
        R2 - 1 ==> R2
        mem[R0] ==> mem[R1] ==> temp
        R0 + size ==> R0
        R1 + size ==> R1
        compare temp with R3. R4 and set F__flag.  M__flag
        according to eeee
            /*  終了条件成立の場合 F__flag=1となる */
        if ( F__flag = 1) then exit
         check__interrupt
until (R2 = 0)

1 ==>  V__flag
```

【０４７７】［ＳＭＯＶ／Ｂのオペレーション］

```
335                                                    336
0 ==>  V__flag
repeat
        R2 - 1 ==> R2
        R0 - size ==> R0
        R1 - size ==> R1
        mem[R0] ==> mem[R1] ==> temp
        compare temp with R3, R4 and set F__flag,  M__flag
        according to eeee
            /*  終了条件成立の場合 F__flag=1となる */
        if ( F__flag = 1) then exit
         check__interrupt
until (R2 = 0)
1 ==>  V__flag
```

ＳＭＯＶでは、Ｒ２の初期値が何であっても、かならず１個以上のエレメントは処理される。ＳＭＯＶの終了要因をまとめると、次のようになる。

１．エレメント（データ）数（Ｒ２）による終了

エレメント数によって命令を終了した場合には、Ｖ＿ｆｌａｇ＝１となる。２のケースとは同時には起こらない。

２．終了条件による終了

この時は、Ｆ＿ｆｌａｇ＝１となる。終了条件を満足したエレメントの転送も行なわれる。

【０４７８】ＳＣＭＰ

ＳＣＭＰでは、エレメント数による命令の終了と終了条件による命令の終了のほかに、比較データの不一致によって命令を終了することがある。ＳＣＭＰでデータの不一致により命令を終了した場合にも、終了条件により命令を終了した場合と同様に、ポインタの更新が行なわれてから命令が終了する。ＳＣＭＰでは、エレメント数による終了要因と他の終了要因とが同時に発生することはないが、終了条件とデータ不一致の終了要因が同時に満たされる可能性はある。ＳＣＭＰがエレメント数によって終了した場合には、次のエレメントの比較は行なわず、次のエレメントが不一致または終了条件成立であっても、Ｖ＿ｆｌａｇ＝０，Ｆ＿ｆｌａｇ＝０，Ｚ＿ｆｌａｇ＝１として命令を終了する。ＳＣＭＰのオペレーションをまとめると、次のようになる。ただし、最終的な結果が同じであれば、メモリアクセスの順番が下記のものと異なっていても構わない。これと等価の動作をすればよい。

【０４７９】［ＳＣＭＰ／Ｆのオペレーション］

```
337                                                    338
0 ==>  V__flag
repeat
        R2 - 1 ==> R2
        mem[R0] ==> temp1
        mem[R1] ==> temp2
        R0 + size ==> R0
        R1 + size ==> R1
        compare temp1 with temp2 and set Z__flag,
         L__flag, X__flag
           /*  データ不一致の場合 Z__flag=0となる */
        compare temp1 with R3, R4 and set  F__flag,  M__flag
        according to eeee
           /*  終了条件成立の場合 F__flag=1となる */
        if ( F__flag = 1 .or.  Z__flag = 0) then exit
            /* 終了条件またはデータ不一致による終了
      */       check__interrupt

until (R2 = 0)                          *【０４８０】［ＳＣＭＰ／Ｂのオペレーション］
1 ==>  V__flag
                                *
    0 ==>  V__flag
    repeat
            R2 - 1 ==> R2
            R0 - size ==> R0
            R1 - size ==> R1
            mem[R0] ==> temp1
            mem[R1] ==> temp2
            compare temp1 with temp2 and set Z__flag,
             L__flag,  X__flag
               /*  データ不一致の場合 Z__flag=0となる */
            compare temp1 with R3, R4 and set  F__flag,  M__flag
            according to eeee
               /*  終了条件成立の場合 F__flag=1となる */
            if ( F__flag = 1 .or.  Z__flag = 0) then exit
                /* 終了条件またはデータ不一致による終了
         */       check__interrupt
    until (R2 = 0)
    1 ==>  V__flag
```

【０４８１】ＳＣＭＰの終了要因をまとめると、次のようになる。

１．エレメント（データ）数（Ｒ２）による終了

この時、Ｚ__ｆｌａｇ＝１，Ｆ__ｆｌａｇ＝０，Ｖ__ｆｌａｇ＝１となる。２，３のケースとは同時には起こらない。

２．終了条件による終了

この時は、Ｆ__ｆｌａｇ＝１となる。終了条件を満足したエレメントの比較も行なわれ、その比較結果がＺ__ｆｌａｇ，Ｌ__ｆｌａｇ，Ｘ__ｆｌａｇに設定される。比

339

較が不一致だった場合は、２，３の２つの終了要因が同時に満たされたことに相当する。

３．比較中のエレメントの不一致による終了

この時は、不一致のあったエレメントの比較結果がＺ＿ｆｌａｇ（＝０），Ｌ＿ｆｌａｇ，Ｘ＿ｆｌａｇに設定される。Ｖ＿ｆｌａｇは０となる。

【０４８２】ＳＳＣＨ　＊

```
0 ==>  V__flag
repeat
        R2 - 1 ==> R2
        mem[R0] ==> temp
        R0 + size ==> R0
        compare temp with R3, R4 and set F__flag,  M__flag
        according to eeee
              /* 終了条件成立の場合 F__flag=1となる */
        if ( F__flag = 1) then exit
              /* 終了条件（検索条件）による終了 */
         check__interrupt
until (R2 = 0)
1 ==>  V__flag
```

【０４８３】［ＳＳＣＨ／Ｒのオペレーション］

```
0 ==>  V__flag
repeat
        R2 - 1 ==> R2
        mem[R0] ==> temp
        R0 + R5 ==> R0
        compare temp with R3, R4 and set F__flag,  M__flag
        according to eeee
              /*  終了条件成立の場合 F__flag=1となる */
        if ( F__flag = 1) then exit
              /*  終了条件（検索条件）による終了 */
         check__interrupt
until (R2 = 0)
1 ==>  V__flag
```

ＳＳＣＨの終了要因をまとめると、次のようになる。

１．エレメント（データ）数（Ｒ２）による終了

この時、Ｖ＿ｆｌａｇ＝１となる。２のケースとは同時には起こらない。

２．終了条件（検索条件）による終了

この時は、Ｆ＿ｆｌａｇ＝１となる。

【０４８４】ＳＳＴＲ

ＳＳＴＲではフラッグ変化は起こらない。ＳＳＴＲのオペレーションをまとめると次のようになる。

［ＳＳＴＲのオペレーション］

340

＊ＳＳＣＨが終了条件（検索条件）によって終了した場合には、／Ｆ，／Ｒとも、命令実行後のポインタは終了条件の成立したエレメントの次のエレメントを指す。また、ＳＳＣＨがエレメント数によって終了した場合にも、命令実行後のポインタは次のエレメントを指す。ＳＳＣＨのオペレーションをまとめると次のようになる。

［ＳＳＣＨ／Ｆのオペレーション］

```
repeat
        R2 - 1 ==> R2
        R3 ==> mem[R1]
        R1 + size ==> R1
         check__interrupt
until (R2 = 0)
```

【０４８５】Ａ１１－３．高機能命令終了時のレジスタ値

ＴＲＯＮＣＨＩＰで高機能命令を実行した場合、命令終

341 342

了時の各レジスタの値は次のようになる。なお、ＲＸｉｎｉｔは命令実行前のレジスタＲＸの値を示す。また、ＲＸｅｎｄは命令実行後のレジスタＲＸの値を示す。

[BVSCH]

/Pの時は、オフセットがR1init～R1init+R2init-1 の範囲をサーチする。

/Bの時は、オフセットがR1init～R1init-R2init+1 の範囲をサーチする。

R2init(width) ≦0 の場合は、 V_flagをセットして命令を終了する。

R1,R2 は不変である。

- サーチが成功した場合

R0(base address): 不変

R1(offset): サーチ結果
見付かったビットのビットオフセット。

R2(width): まだサーチしていないビットフィールドと見付かったビットとを合わせたビットフィールドの長さ
つまり、/FであればR2init+R1init-R1end.
/BであればR2init-R1init+R1end となる。

- サーチが失敗した場合

R0(base address): 不変

R1(offset): 最後にサーチサーチしたビットの次のビットオフセット。
つまり、/FであればR1init+R2init.
/BであればR1init-R2init となる。
これはBSCHと同じ考え方である。

R2(width): 0

【０４８６】

343 344

[BVMAP]

[BVCPY]

/Fの時は、R1init～R1init+R2init-1 のビットオフセットを持つ領域がsrc. R4init ～R4init+R2init-1 のビットオフセットを持つ領域がdestとなる。

/Bの時は、R1init～R1init-R2init+1 のビットオフセットを持つ領域がsrc.R4init-1～R4init-R2init のビットオフセットを持つ領域がdestとなる。

R2init(width)≦0 の場合は、何もせずに命令を終了する。

R1.R2.R4 は不変である。

R0(src base):	不変
R1(src offset):	/FであればR1init+ R1init./B であればR1init-R2init となる。
R2(width):	0
R3(dest base)	不変
R4(dest offset)	/FであればR4init+R2init./BであればR4init-R2init となる。
R5（演算の種類）	不変（BVMAP のみ）

【０４８７】

[BVPAT]

R4init ～R4init+R2init-1 のビットオフセットを持つ領域がdestとなる。

R2init(width)≦0 の場合は、何もせずに命令を終了する。

R2.R4 は不変である。

R0(pattern):	不変
R2(width):	0
R3(dest base):	不変
R4(dest offset):	R4init+R2init
R5（演算の種類）	不変

【０４８８】［SMOV］

345 346

/F の時は、

ROinit ～ROinit+R2init*エレメントサイズ-1のアドレスを持つ領域がsrc 、

R1init ～R1init+R2init*エレメントサイズ-1のアドレスを持つ領域がdestとなる。

/B の時は、

ROinit-1 ～ROinit-R2init

*エレメントサイズのアドレスを持つ領域がsrc 、

R1init-1 ～R1init-R2init

* エレメントサイズのアドレスを持つ領域がdestとなる。

例えば、H'0000～H'00ffのストリングをH'0300～H'03ffに転送する場合、SMOV/F.Wでコピーすると、

R0=H'0000, R1=H'0300, R2=H'0040

SMOV/B.Wでコピーすると、

R0=H'0100, R1=H'0400, R2=H'0040

とすればよい。ただし、終了条件が成立した場合には、　　もｄｅｓｔ側に転送される。処理が途中で打ち切られる。終了条件の成立したデータ　20　【０４８９】ーエレメント数によって終了した場合(V__flag=1)

R0(src address): /Fであれば ROinit+R2init*エレメントサイズ、
/Bであれば ROinit-R2init*エレメントサイズ

R1(dest address): /Fであれば R1init+R2init*エレメントサイズ、
/Bであれば R1init-R2init*エレメントサイズ

R2(エレメント数):0

R3 (終了条件1) :　不変

R4 (終了条件2) :　不変

- 終了条件が成立して終了した場合(F__flag=1)

R0(src address): /Fの時終了条件の成立したsrc のエレメントの次のエレメントのアドレス
/Bの時終了条件の成立したsrc のエレメントのアドレス

R1(dest address): /Fの時終了条件の成立したsrc のエレメントの次のエレメントを転送すべきdestのアドレス
/Bの時終了条件の成立したsrc のエレメントを転送すべきdestのアドレス
/F, /B ともR1init+R0end-R0init となる。

R2(エレメント数): まだ転送されていないエレメントの数
/F の時R2init-(R0end-R0init)/エレメントサイズ、
/Bの時R2init-(R0init-R0end)/エレメントサイズとなる。

R3 (終了条件1) :　不変

R4 (終了条件2) :　不変

【０４９０】［ＳＣＭＰ］

347　　*348*

/F の時は、

R0init ～R0init+R2init*エレメントサイズ-1のアドレスを持つ領域がsrc1、R1init ～R1init+R2init*エレメントサイズ-1のアドレスを持つ領域がsrc2となる。

/B の時は、

R0init-1 ～R0init-R2init * エレメントサイズのアドレスを持つ領域がsrc1、R1init-1～R1init-R2init * エレメントサイズのアドレスを持つ領域がsrc2となる。

例えば、H'0000～H'00ffのストリングをH'0300～H'03ffのストリングとを比較する場合、

SCMP/F.W で比較すると、

R0=H'0000.R1=H'0300.R2=H'0040

SCMP/B.W で比較すると、

R0=H'0100.R1=H'0400.R2=H'0040

とすればよい。ただし、終了条件が成立した場合には、処理が途中で打ち切られる。終了条件の成立したエレメ*ントについても比較が行なわれ、その結果がＬ＿ｆｌａｇ，Ｘ＿ｆｌａｇ，Ｚ＿ｆｌａｇにセットされる。また、比較中に不一致のエレメントが見付かった場合にも、処理が途中で打ち切られる。

*20

【０４９１】ーエレメント数によって終了した場合

(V__flag=1)

R0(src1 address): /Fであれば R0init+R2init*エレメントサイズ、
/Bであれば R0init-R2init*エレメントサイズ
ただし、R2init＜0 の場合には無変化

R1(src2 address): /Fであれば R1init+R2init*エレメントサイズ、
/Bであれば R1init-R2init*エレメントサイズ

R2(エレメント数): 0

R3（終了条件1 ）:　不変

R4（終了条件2 ）:　不変

ー終了条件の成立、またはエレメント値の不一致によって終了した場合

(F__flag=1 .or.　Z__flag=0)

R0(src1 address): /Fの時終了条件の成立した（または不一致であった）src1のエレメントの次のエレメントのアドレス
/Bの時終了条件の成立した（または不一致であった）src1のエレメントのアドレス

R1(src2 address): /Fの時終了条件の成立した（または不一致であった）src1のエレメントの次のエレメントと対応するsrc2のエレメントのアドレス
/Bの時終了条件の成立した（または不一致であった）src1のエレメントと対応するsrc2のエレメントのアドレス

349 350

/F,/B ともR1init+R0end-R0init となる。

R2(エレメント数): まだ比較されていないエレメントの数
/Fの時R2init-(R0end-R0init)/エレメントサイズ、
/Bの時R2init-(R0init-R0end)/エレメントサイズ
となる。

R3（終了条件1 ）： 不変

R4（終了条件2 ）： 不変

【０４９２】［ＳＳＣＨ］ 10

/F の時は、

R0init ～R0init+R2init*エレメントサイズ-1のアドレスを持つ領域を検索する。

/R の時は、

R0init ～R0init+R5*R2init-1のアドレスを持つ領域を、R5毎に飛び飛びに検索する。

ただし、終了条件（検索条件）が成立した場合には、処理が途中で打ち切られる。

－エレメント数によって終了した場合 （Ｖ＿ｆｌａｇ ＝１）

R0(src address): /Fであれば R0init+R2init*エレメントサイズ、
/Rであれば R0init+R2init*R5

R2(エレメント数): 0

R3（終了条件1 ）： 不変

R4（終了条件2 ）： 不変

R5(ポインタ 更新値): 不変

－終了条件（検索条件）が成立して終了した場合（Ｆ＿ ｆｌａｇ＝１）

R0(src address): 終了条件の成立したsrc のエレメントのアドレス

R2(エレメント数): 終了条件の成立したエレメントと、まだ検索されていないエレメントの合計数
/Fの時R2init-(R0end-R0init)/エレメントサイズ、
/Rの時R2init-(R0end-R0init)/R5となる。

R3（終了条件1 ）： 不変

R4（終了条件2 ）： 不変

R5(ポインタ 更新値): 不変

【０４９３】［ＳＳＴＲ］Ｒ１ｉｎｉｔ～Ｒ１ｉｎｉｔ＋Ｒ２ｉｎｉｔ＊エレメントサイズ－１のアドレスを持つ領域に、Ｒ３で指定されるデータを繰り返し書き込む。他のストリング命令とは異なり、終了条件の指定は行なわない。また、フラッグのセットも行なわない。Ｒ２ｉｎｉｔ（ｗｉｄｔｈ）≦０の場合は、即座に命令を終了する。Ｒ１，Ｒ２は不変である。

R1(dest address): R1init+R2init*エレメントサイズ

R2(エレメント数): 0

R3(書き込みﾃﾞｰﾀ): 不変

【０４９４】［ＱＳＣＨ］

－キュー終了値（Ｒ２）によって終了した場合（Ｖ＿ｆｌａｇ＝１）

351 352

R0(entry address):R2init

R1(previous entry):R0endで示されるエントリの直前（/Fの場合）または直後
(/B の場合）のエントリのアドレス

R2(キュー終了値): 不変

R3（終了条件1 ）： 不変

R4（終了条件2 ）： 不変

R5（オフセット）：不変

R6（マスク）： 不変

－終了条件（検索条件）が成立して終了した場合（F__ ｆｌａｇ＝1）

R0(entry address):終了条件の成立しているキューエントリのアドレス

R1(privious entry):R0endで示されるエントリの直前（/Fの場合）または直後
(/Bの場合）のエントリのアドレス

R2(キュー終了値): 不変

R3（終了条件1 ）： 不変

R4（終了条件2 ）： 不変

R5（オフセット）：不変

R6（マスク）： 不変

【０４９５】付録１２．オペランドが干渉した場合の動作

一つの命令が複数のオペランドを持ち、その中で、@ＳＰ＋，@－ＳＰモードとＳＰを参照するモードとを併用した場合には、@ＳＰ＋，@－ＳＰモードによって更新されたＳＰ値がいつから反映されるかということが問題になる。この問題をより一般的に考えると、オペランドが干渉していた場合の動作を明確に規定すればよいということになる。この資料は、本発明装置で、命令中のオペランドが干渉を起こした場合に、その動作をきちんと定義することを目的としたものである。資料の最初の方でこの問題に対する考え方を述べ、資料の後半では、実際に本発明装置の動作の規定を行っている。

【０４９６】Ａ１２－１．命令実行モデル

オペランド干渉の問題を整理するために、まず、各命令からメモリやレジスタに対して行われるｒｅａｄ，ｗｒｉｔｅ，ｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅの基本アクセス操作をすべて含むような仮想的な命令パターンを考え、それを以下のようにモデル化する。

RA1. 第一readオペランド(RO1) 実効アドレス計算

↓

RR1. 第一readオペランド(RO1) 読みだし

↓

RA2. 第二readオペランド(RO2) 実効アドレス計算

↓

RR2. 第二readオペランド(RO2) 読みだし

353　　354

↓

RA3.　第三readオペランド(RO3) 実効アドレス計算

↓

RR3.　第三readオペランド(RO3) 読みだし

↓

AA1.　第一address オペランド(AO1) 実効アドレス計算

↓

AA2.　第一address オペランド(AO2) 実効アドレス計算

↓

WA1.　第一write オペランド(WO1) 実効アドレス計算

↓

MA1.　第一read-modify-write オペランド(MO1) 実効アドレス計算

MR1.　第一read-modify-write オペランド(MO1) 読みだし

↓

EX1 ～n.命令の実行(1～n)

↓

MW1.　第一read-modify-write オペランド(MO1) 書き込み

↓

WW1.　第一write オペランド(WO1) 書き込み

【０４９７】・モデルを構成するＲＡ１～ＷＷ１の夫々の要素を「ステップ」と呼ぶ。

・@ＳＰ＋，@－ＳＰのアドレッシングモードを指定した場合、ＳＰ値の更新は「実効アドレス計算」のステップ（ＲＡ１，ＲＡ２，ＲＡ３，ＭＡ１，ＷＡ１）で行われるものとする。

・各ステップは、次のいずれかのパターンでリソースをアクセスする。ただし、「リソース」とはメモリまたはプログラムから見えるレジスタを指す。

－リソースアクセスなし

－－つまたは複数のリソースからのｒｅａｄ

－－つのリソースへのｗｒｉｔｅ（ＥＸ１～ＥＸｎ，ＭＷ１，ＷＷ１のみ）

－リソースＳＰのｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅ（ＲＡ１，ＲＡ２，ＲＡ３，ＭＡ１，ＷＡ１，ＷＡ２）

・各ステップでリソースの値の更新があった場合には、次のステップから更新された値が反映される。

・一般の命令では、このうち一部のステップのみが存在する。

・ＥＸ１～ＥＸｎのステップは、実際には、命令の種類によってＥＸ１，ＥＸ２，ＥＸ３・・・のいくつかのステップに分かれている。

上記のモデルでは、各ステップで書き込みを行なうリソースが高々一つであること、ＳＰのｒｅａｄ－ｍｏｄｉｆｙ－ｗｒｉｔｅのケースを除けば、一つのステップ内でｒｅａｄとｗｒｉｔｅが混在するケースはないこと、リソースに書き込んだ値が次のステップから反映されること、といった条件を明らかにしているので、ステップ間で共通のオペランドをアクセスした場合にも動作は明確に規定される。したがって、各命令の動作を上記のモデルにうまく当てはめることができれば、オペランドが重なっていた場合の動作もきちんと規定することができる。本発明装置のほとんどの命令の動作は、上記の実行パターンのサブセットとして成り立っている。ただしどの動作がどのステップに対応するかということについては、命令動作の意味付けによっていろいろな解釈ができる。例えばＡＣＢ：Ｇ　@ＳＰ＋，ＳＰ，@ＳＰ＋，ｎｅｗｐｃの場合には、次の①のような解釈をするのが自然である。つまり、ＡＣＢ命令の各動作ステップやオペランドと、上記のモデルとの対応は、次の①のように行なうのが普通であり、実際、これが正しいＴＲＯＮ仕様となっている。

【０４９８】①ＡＣＢのＳＴＥＰオペランドがモデルのＲ０１に対応

ＡＣＢのＬＩＭＩＴオペランドがモデルのＲ０２に対応

ｘｒｅｇオペランドはＥＸ１～ＥＸ３の中だけで扱い、Ｒ０ｎ，Ｍ０ｎ，Ｗ０ｎとは考えない。

RA1.　SP==>roladdr; SP+size==>SP

RR1.　@roladdr==>step

RA2.　SP==>ro2addr; SP+size==>SP

RR2.　@ro2addr==>limit

RA3 ～MR1.　なし

355

EX1.　SP==>xreg

EX2.　xreg+step==>xreg

　　if(xreg<limit)jump to newpc endif

EX3.　xreg==>SP

MW1 ～WW1.　なし

*

step:　@(initSP)

xreg初期値:　initSP+size*2

limit:　@(initSP+size)

EXの後のxreg:　initSP+size*2+@(initSP)

ジャンプ条件:　initSP+size*2+@(initSP) <@(initSP+size)

SP最終値:　initSP+size*2+@(initSP)

しかし、実際のＡＣＢ命令の動作とモデルとの対応を次の②のようにすることも不可能ではない。②の動作は、厳密には①と同じではない。

【０４９９】②ＡＣＢのｓｔｅｐオペランドがモデルのＲ０１に対応

ＡＣＢのｘｒｅｇオペランドが読みだし時Ｒ０２に、書き込み時Ｗ０１に対応ＡＣＢのｌｉｍｉｔオペランドがモデルのＲ０３に対応

RA1.　SP==>roladdr; SP+size==>SP

RR1.　@roladdr==>step

RA2.　何も参照しない

RR2.　SP==>xreg

RA3.　SP==>ro3addr; SP+size==>SP

RR3.　@ro3addr==>limit

AA1 ～MR1.　なし

EX.　xreg+step==>xreg

　　if(xreg<limit) jump to newpc endif

※

step:　@(initSP)

xreg初期値:　initSP+size

limit:　@(initSP+size)

EXの後のxreg:　initSP+size+@(initSP)

ジャンプ条件:　initSP+size+@(initSP) <@(initSP+size)

SP最終値:　initSP+size+@(initSP)

これは①とは異なる動作である。仕様書の命令の解説の項だけでは①が正しいか②が正しいかは厳密にはわからないため、命令によっては、モデルとの対応付けの曖昧な場合が出てくる。モデルとの対応付けが違うと、上の例のように細かい動作が異なってくることがある。この資料では本発明装置の命令について、上記のような方法で命令動作とモデルとの対応を決め手いくことにより、オペランドが重なった場合の動作を明確に規定することを目的としている。

【０５００】Ａ１２－２．基本原則

命令について個別に検討する前に、オペランドの干渉に

356

＊－ｒｏ１ａｄｄｒ，ｒｏ２ａｄｄｒ，ｓｔｅｐ，ｌｉｍｉｔ　ｘｒｅｇは内部変数であり、モデルで使用する意味での「リソース」には含めないこの場合、命令実行前のＳＰ値をｉｎｉｔＳＰとすると、

※

MW1.　なし

WW1.　xreg==>SP

－ｒｏ１ａｄｄｒ，ｒｏ３ａｄｄｒ，ｓｔｅｐ　ｌｉｍｉｔ，ｘｒｅｇは内部変数であり、モデルで使用する意味での「リソース」には含めないこの場合、命令実行前のＳＰ値をｉｎｉｔＳＰとすると、

ついての基本原則を述べる。

（原則１）短縮形と一般形では、全く同じ動作をしなければならない。すなわち、命令フォーマットの違いはオペランド干渉の問題には影響しない。短縮形は完全に一般形のサブセットとなるべきであり、動作の異なる短縮形は混乱を招く。命令実行モデルは、インプリメントへの依存性やインプリメントの都合を考えたものではないため、場合によっては、規定した通りにインプリメントできないことがある。例えば、ＡＤＤ：Ｌ　＠ＳＰ＋，ＳＰがＡＤＤ：Ｇ　＠ＳＰ＋，ＳＰと異なった動作をする場合があるというのは、この例である。このようなケ

357

ースに関しては、命令実行モデルの方はそのままにしておき、場合に応じて例外として扱う。

（原則２）基本的には、命令ビットパターンの中でのオペランドの出現順序にしたがって、命令のオペランドとモデルでの第一ｒｅａｄオペランド（Ｒ０１）、第二ｒｅａｄオペランド（Ｒ０２）・・・との対応を決める。こうしておけば、一般にはインプリメント方法とも矛盾しない。命令実行モデルは、各命令の動作とビットパターンから考えて、最も自然な形になるようにする。短縮形と一般形を持つ命令の場合、命令実行モデルと実際の命令動作との対応が自然になるかどうかは、どのフォーマットの命令ビットパターンを基準にするかによって異なるが、この場合は減速として一般形を基準にする。命令動作とモデルとの対応という点に関しては、命令ビットパターンの都合は考えるが、インプリメントの都合は特に考えない。

（原則３）原則２があっても、汎用アドレッシングでないオペランド（ＡＣＢのｘｒｅｇなど）については、モデルでの第ｎ－ｒｅａｄオペランド（Ｒ０ｎ）、第ｍ－ｗｒｉｔｅオペランド（Ｗ０ｍ）としては扱わず、ＥＸのステップ中だけで扱うのが普通である。

（原則４）また、命令の意味の中に暗黙に含まれているメモリアクセス（ＰＵＳＨ，ＰＯＰでのスタック操作など）についても、モデルでの第ｎ－ｒｅａｄオペランド（Ｒ０ｎ）、第ｍ－ｗｒｉｔｅオペランド（Ｗ０ｍ）としては扱わず、ＥＸのステップ中だけで扱うのが普通である。

さらに、各命令ごとに特殊な事情のある場合や、多数のオペランドを持つ場合（ＬＤＭ，ＳＴＭなど）には、これらの原則が当てはまらないことがある。詳しくは命令ごとに検討する。

【０５０１】Ａ１２－３．各命令での実際

本発明装置の命令をいくつかのパターンに分類してモデルとの対応付けを行ない、オペランドが干渉した場合の動作を明確にする。（原則１）により、命令フォーマットが違ってもオペランドが干渉した場合の動作は変わらないので、命令フォーマットの違いによる場合分けを行なう必要はない。なお、動作例の中に出てくるｉｎｉｔＳＰは、命令実行前のＳＰの意味である。また、動作例で出てくる命令のオペランドサイズは、断わりのない限りすべて３２ビットとする。’＝＝＞’は値の代入を、’〔～〕’はコメントを示す。ＲＡ１～ＷＷ１のステップのうち、その命令で使用していないステップについては、説明を省略してある。

【０５０２】Ｏオペランド（Ｎ）

```
PUSH    src
LDPSB   src
LDPSM   src
JRNG    vector
```

358

該当命令：

ＮＯＰ
ＰＩＢ
ＲＴＳ
ＲＲＮＧ
ＴＲＡＰ
ＲＥＩＴ
ＳＴＣＴＸ
ＰＴＬＢ
ＢＶＳＣＨ
ＢＶＭＡＰ
ＢＶＣＰＶ
ＢＶＰＡＴ
ＳＭＯＶ
ＳＣＭＰ
ＳＳＣＨ
ＳＳＴＲ
ＱＳＣＨ

命令の動作は、すべてモデルのＥＸのステップだけで行なわれ、それ以外の部分は存在しない。したかって、オペランドの干渉については特に問題とはならない。高機能命令では、ＥＸのステップ数が有限とはならない場合も生じるため、このモデルでは正確に表現できないことがある。しかし、これはオペランドの干渉の問題には直接関係しないので、ここでは議論しない。

〔命令実行モデルとオペランドとの対応〕

ＥＸ．　命令固有の動作

【０５０３】１オペランドイミディエート（Ｉ）

当該命令：

```
BRA     newpc
Bcc     newpc
BSR     newpc
TRAPA   vector
WAIT    imask
```

オペランドを一つもつが、オペランドの値は命令コード中で直接指定されるため、オペランドの干渉については問題とならない。ｎｅｗｐｃ，ｖｅｃｔｏｒ，ｉｍａｓｋはＥＸのステップで操作すると考える。

〔命令実行モデルとオペランドとの対応〕

ＥＸ．　命令固有の動作（ｎｅｗｐｃ，ｖｅｃｔｏｒ，ｉｍａｓｋの参照を含む）

【０５０４】１オペランドｒｅａｄ（Ｒ）

該当命令：

359

このパターンの命令の場合、ｓｒｃ，ｖｅｃｔｏｒをモデルの第一ｒｅａｄオペランド（Ｒ０１）として扱う。

ｓｒｃ＝@ＳＰ＋の場合、命令固有の動作を行なう時点＊

```
RA1.    src, vector のアドレス計算==>roladdr
RR1.    @roladdr==>src, vector
EX.     命令固有の動作（命令による暗黙のスタック操作を含む）
        PUSHでは    src==>PSB
        LDPSB では  .op.(PSB, src)==>PSB
        LDPSM では  .op.(PSM, src)==>PSM
```

例えばＰＵＳＨ@ＳＰ＋にこのモデルを適用すると、ＥＸでＰＵＳＨ命令固有の動作（ｓｒｃ＝＝＞@－ＳＰ）を行なう前に、ＲＡ１．で既にｓｒｃ－@ＳＰ＋によるＳＰの更新が行われることがわかる。ただし、間違いを防ぐため、ＰＵＳＨ@ＳＰ＋は禁止（ＲＩＥ）となっている（図３９６参照）。

【０５０５】１オペランドａｄｄｒｅｓｓ（Ａ）

該当命令：

```
PUSHA   srcaddr
PSTLB   prgaddr
LDCTX   ctxaddr
ACS     chkaddr
JMP     newpc
JSR     newpc
```

20

※

```
AA1.    srcaddr, prgaddr, ctxaddr, chkaddr, newpc のアドレス計算
        ==>aoladdr
EX.     命令固有の動作（命令による暗黙のスタック操作を含む）
        PUSHA では  aoladdr==>@-SP
        JSR では    PC==> @-SP; aoladdr ==>PC
```

ＰＵＳＨＡ　@ＳＰとＪＳＲ　@ＳＰの動作は図３９７のとおり。

【０５０６】１オペランドｗｒｉｔｅ（Ｗ）

該当命令：

```
POP     dest
STPSB   dest
STPSM   dest
```

★40

```
WA1.    destの実効アドレス計算==>woladdr
EX.     命令固有の動作（命令による暗黙のスタック操作を含む）
        POP では    @SP+==> dest2
        STPSB では  .op.(PSB)==>dest2
        STPSM では  .op.(PSM)==>dest2
WW1.    dest2==>@woladdr
```

例えばＰＯＰ　@－ＳＰにこのモデルを適用すると、ＥＸ．でＰＯＰ命令固有の動作（@ＳＰ＋＝＝）ｄｅｓｔ２）を行なう前に、ＷＡ１．で既にｄｅｓｔ＝@－ＳＰによるＳＰの更新が行なわれることがわかる。また、ＰＯＰ　@（ｄ，ＳＰ）にこのモデルを適用すると、ＥＸ．でＰＯＰ命令固有の動作（@ＳＰ＋＝＝）ｄｅｓｔ

360

＊では、既にＳＰのインクリメントが行なわれていることになる。

〔命令実行モデルとオペランドとの対応〕

※ｓｒｃａｄｄｒ，ｐｒｇａｄｄｒ，ｃｔｘａｄｄｒ，ｃｈｋａｄｄｒ，ｎｅｗｐｃをモデルの第一ａｄｄｒｅｓｓオペランド（Ａ０１）として扱う。なお、ＡＡ１では@ＳＰ＋，@－ＳＰのモードは利用できない。

〔命令実行モデルとオペランドとの対応〕

★ｄｅｓｔを第一ｗｒｉｔｅオペランド（Ｗ０１）として扱う。この場合、ＥＸのステップより前のＷＡ１のステップでｄｅｓｔの実効アドレス計算が行なわれ、値の書き込みのみＥＸより後のＷＷ１のステップで行なわれることになる。

〔命令実行モデルとオペランドとの対応〕

50

361

２）を行なう前に、ＷＡ１．でｄｅｓｔ＝@（ｄ，ＳＰ）のアドレス計算が行なわれ、ｄｅｓｔのアドレス計算にはｉｎｉｔＳＰが使用されることがわかる。ただし、間違いを防ぐため、実際にはＰＯＰ　@－ＳＰは禁止（ＲＩＥ）となっている。また、ＰＯＰ　@（ｄ，ＳＰ）については、@（ｄ，Ｒｎ）のアドレッシングモードでＲｎ＝ＳＰの時のみ実行禁止とするのは無理があること、スタック操作においてＰＯＰ　@（ｄｉｓｐ，ＳＰ）を使用する用途も考えられること、により、禁止にはしていない。ＰＯＰ命令の動作例は図３９８のようになる。

【０５０７】１オペランドｒｍｗ（Ｍ）

該当命令：

```
NEG     dest
NOT     dest
SHXL    dest
SHXR    dest
```

ｄｅｓｔをＭ０１に対応させる。この場合、ＭＡ１で@ＳＰ＋，@－ＳＰを指定することはできない。

〔命令実行モデルとオペランドとの対応〕

```
MA1.    destの実効アドレス計算==>moladdr
MR1.    @moladdr==> dest1
EX.     命令固有の動作
            .op.(dest1) ==> dest2
MW1.    dest2 ==> @moladdr
```

【０５０８】２オペランドｒｅａｄ～ｗｒｉｔｅ（ＲＷ）

該当命令：

```
MOV     src,  dest
MOVU    src,  dest
RVBY    src,  dest
RVBI    src,  dest
PACKss  src,  dest
UNPKss  src,  dest, adj
```

ｓｒｃをＲ０１、ｄｅｓｔをＷ０１に対応させる。した＊

```
AA1.    src, srcaddr, queue の実効アドレス計算==> aoladdr
WA1.    destの実効アドレス計算==>woladdr
EX.     命令固有の動作
            .op.(aoladdr)==>dest2
WW1.    dest2 ==>@woladdr
```

ＳＴＣ，ＭＯＶＡの動作例との対応は図４０１のとおり。ｍｏｖａ　@ＳＰ，@－ＳＰの場合、ｓｒｃａｄｄｒの実効アドレス計算（ＡＡ１）のステップで既にＳＰが参照されており、ｄｅｓｔの実効アドレス計算（ＷＡ１）ステップにおけるＳＰの更新はｓｒｃａｄｄｒに反

362

＊がって、ｓｒｃ側の実効アドレス計算とそれに伴うＳＰの更新がすべて終わってからｓｒｃのフエッチを行ない、その後ｄｅｓｔ側の実効アドレス計算を行ない、最後に命令固有の動作を行って結果をｄｅｓｔにセットする。ＵＮＰＫｓｓのａｄｊはＥＸのステップで扱い、第ｎ－ｒｅａｄオペランドとはしない。

〔命令実行モデルとオペランドとの対応〕

```
RA1.    src の実効アドレス計算==> roladdr
RR1.    @roladdr==> src1
WA1.    destの実効アドレス計算==> woladdr
EX.     命令固有の動作
            .op.(src1)==> dest2
            UNPKssではadj を参照
WW1.    dest2 ==>@woladdr
```

ＭＯＶの動作例は図３９９のとおり。ＰＡＣＫ，ＵＮＰＫの動作例は図４００のとおり。

【０５０９】２オペランドａｄｄｒｅｓｓ～ｗｒｉｔｅ（ＡＷ）

該当命令：

```
STC     src, dest
STP     src, dest
MOVA    srcaddr, dest
MOVPA   srcaddr, dest
STATE   srcaddr, dest
QDEL    queue, dest
```

制御空間、物理空間など特殊な空間からの読みだしを行なう命令では、特殊空間側の実効アドレスｓｒｃ，ｓｒｃａｄｄｒをＡ０１として扱い、特殊空間の実際のアクセスはＥＸのステップで行なう。また、ＱＤＥＬ命令でも、キューエントリを指定する側の実効アドレスｑｕｅｕｅをＡ０１として扱い、キューリンクの実際の操作はＥＸのステップで行なう。こうしておけば、命令ビットパターンやインプリメント方法ともうまく対応する。ＡＡ１では、@ＳＰ＋，@－ＳＰを指定することはできない。ｄｅｓｔはＷ０１として扱う。

映されない。これに対して、ＳＴＣ　@ｓｐ０，@－ＳＰがモデル通りの動作をするものと考えると、ｓｒｃの実効アドレス計算（ＡＡ１）のステップではまだＳＰが参照されておらず、制御レジスタとしてのＳＰのアドレス（Ｈ’０１２４）が計算されるだけである。ＳＰが参照されるのはＥＸのステップであり、これはｄｅｓｔの実効アドレス計算（ＷＡ１）ステップにおいてＳＰを更新した後である。したがって、ｉｎｉｔＳＰ－４がｄｅｓｔに転送されることになる。ただし、ＳＴＣ　@ｓｐ

363

０，@－ＳＰの動作については、以上のようなモデル通りの動作をするかどうかがインプリメント依存になっている。つまり、モデル通りｉｎｉｔＳＰ－４を転送する場合と、モデルに対しては例外の扱いとなるが、ｉｎｉｔＳＰを転送する場合とがある。

【０５１０】２オペランドｒｅａｄ～ａｄｄｒｅｓｓ（ＲＡ）

該当命令：

```
LDC     src, dest
LDP     src, dest
LDATE   src, destaddr
BTST    offset, base
BSET    offset, base
BCLR    offset, base
BNOT    offset, base
BSETI   offset, base
BCLRI   offset, base
BFEXT   offset, width, base, dest
BFEXTU  offset, width, base, dest
BFINS   src, offset, width, base
BFINSU  src, offset, width, base
BFCMP   src, offset, width, base
BFCMPU  src, offset, width, base
CHK     bound, index, xreg
```

【０５１１】制御空間、物理空間など特殊な空間への書き込みを行なう命令ＬＤＣ，ＬＤＰ，ＬＤＡＴＥでは、特殊空間側の実効アドレスｄｅｓｔ，ｄｅｓｔａｄｄｒをＡ０１、ｓｒｃをＲ０１として扱う。特殊空間の実際のアクセスはＥＸで行なう。ビット操作命令では、ｂａｓｅの実効アドレスをＡ０１、ｏｆｆｓｅｔをＲ０１として扱う。ｂａｓｅとｏｆｆｓｅｔの合成を行ない、操作対象となるビットを実際にアクセスするのは、ＥＸの中で行なわれる。固定長ビットフィールド操作命令でも、ビット操作命令と同様に、ｂａｓｅの実効アドレスをＡ０１、ｏｆｆｓｅｔをＲ０１として扱う。ｏｆｆｓｅｔとｂａｓｅ以外のオペランド、即ちｗｉｄｔｈ，ｓｒｃ，ｄｅｓｔについては、ＥＸの中でのみアクセスし、第ｎ－ｒｅａｄオペランド（Ｒ０ｎ）、第ｍ－ｗｒｉｔｅオペランド（Ｗ０ｍ）としては扱わない。ＣＨＫの場合は、ビットパターンとの関係から、ｉｎｄｅｘをＲ０１、ｂｏｕｎｄをＡ０１として扱い、ｂｏｕｎｄの内容である（上限値：下限値）の読みだしとｘｒｅｇへの書き込みはＥＸで行なうことにする。以上のような形で各命令の動作とモデルとの対応をつけておけば、命令ビットパターンやインプリメント方法とも矛盾しない。ＡＡ１では、@ＳＰ＋，@－ＳＰを指定することはできない。ｓｒｃ，ｏｆｆｓｅｔ，ｉｎｄｅｘ＝@ＳＰ＋の場合、ｄｅｓｔ，ｄｅｓｔａｄｄｒ，ｂａｓｅ，ｂｏｕｎｄの実効アドレス計算で参照されるＳＰ値としては、インクリメント後の値が使用されることになる。ＣＨＫ命令の場合、これはアセンブラ表記での順序と逆になっているので、注意が必要である。また、固定長ビットフィールド命令でｏｆｆｓｅｔ＝@ＳＰ＋を指定し、ｓｒｃ，ｗｉｄｔｈなどで同じＳＰを指定した場合のＳＰ値としては、やはりインクリメント後の値が使用されることになる。アセンブラ表記では、ＢＦＩＮＳ，ＢＦＣＭＰの場合にｏｆｆｓｅｔよりもｓｒｃの方が先に書かれるようになっているが、実際にはｏｆｆｓｅｔの実効アドレス計算で更新されたＳＰがｓｒｃに反映されるので、注意が必要である。

364

【０５１２】

```
RA1.    src, offset, indexの実効アドレス計算==> roladdr
RR1.    @roladdr==> src1
AA1.    dest, destaddr, base, bound
        の実効アドレス計算==> aoladdr
EX.     命令固有の動作
            .op.(src1, aoladdr)
          CHK では
            .op.(src1, aoladdr)==>xreg
          BFINS, BFINSU, BFCMP, BFCMPU では、
            .op.(src1, aoladdr, width, src)
          BFEXT, BFEXTU では、
            .op.(src1, aoladdr, width)==> dest
```

BFINS，BFEXTの動作例は図４０２のとおり。

【０５１３】２オペランドｒｅａｄ～ｒｅａｄ（ＲＲ）

該当命令：

365

```
CMP     src1, src2
CMPU    src1, src2
INDEX   indexsize, subscript, xreg
ACB     step, xreg, limit, newpc
SCB     step, xreg, limit, newpc
```

ＣＭＰ，ＣＭＰＵでは、ｓｒｃ１をＲ０１に、ｓｒｃ２をＲ０２に対応させる。ＩＮＤＥＸでは、ｉｎｄｅｘｓｉｚｅをＲ０１に、ｓｕｂｓｃｒｉｐｔをＲ０２に対応させる。ｘｒｅｇはＥＸで扱う。ＡＣＢ，ＳＣＢではｓｔｅｐをＲ０１に、ｌｉｍｉｔをＲ０２に対応させる。＊

366

＊ｘｒｅｇ，ｎｅｗｐｃはＥＸで扱う。したがって、ｓｒｃ１，ｉｎｄｅｘｓｉｚｅ，ｓｔｅｐ＝@ＳＰ＋の場合、ｓｒｃ２，ｓｕｂｓｃｒｉｐｔ，ｌｉｍｉｔの実効アドレス計算で参照されるＳＰとしては、インクリメント後の値が使用される。また、ＩＮＤＥＸ，ＡＣＢ，ＳＣＢ命令でｉｎｄｅｘｓｉｚｅ，ｓｔｅｐ＝@ＳＰ＋またはｓｕｂｓｃｒｉｐｔ，ｌｉｍｉｔ＝@ＳＰ＋の場合、ｘｒｅｇとして参照されるＳＰとしては、やはりインクリメント後の値が使用される。

【０５１４】

```
RA1.    src1,indexsize,step のアドレス計算==> roladdr
RR1.    @roladdr==> src11
RA2.    src2,subscript,limitの実効アドレス計算==> ro2addr
RR2.    @ro2addr==> src21
EX.     命令固有の動作
            .op.(src11, src21)
        INDEX では
            .op.(src11, src21,xreg) ==> xreg
        ACB, SCBでは
            .op.(src11, src21,xreg, newpc)==> xreg
```

ＣＭＰの動作例は図４０３のとおり。ＩＮＤＥＸの動作例は図４０４のとおり。ただし、ＩＮＤＥＸ命令については、インプリメントの制約によって、ｓｕｂｓｃｒｉｐｔ＝@ＳＰ＋，ｘｒｅｇ＝ＳＰの場合に必ずしもモデル通りの動作ができない場合がある。詳しくは２オペランドｒｅａｄ～ｒｍｗ（ＲＭ）の項を参照。ＡＣＢ，ＳＣＢの動作例は図４０５のとおり。

【０５１５】２オペランドｒｅａｄ～ｒｍｗ（ＲＭ）

該当命令：

```
ADD     src,  dest
ADDU    src,  dest
ADDX    src,  dest
SUB     src,  dest
SUBU    src,  dest
SUBX    src,  dest
MUL     src,  dest
MULU    src,  dest
DIV     src,  dest
DIVU    src,  dest
REM     src,  dest
REMU    src,  dest
AND     src,  dest
OR      src,  dest
XOR     src,  dest
SHA     count, dest
SHL     count, dest
ROT     count, dest
ADDDX   src,  dest
SUBDX   src,  dest
BSCH    data, offset
MULX    src, dest, tmp
DIVX    src, dest, tmp
```

ｓｒｃ，ｃｏｕｎｔ，ｄａｔａをＲ０１に、ｄｅｓｔ，ｏｆｆｓｅｔをＭ０１に対応させる。ｓｒｃ＝@ＳＰ＋の場合、ｄｅｓｔの実効アドレス計算で参照されるＳＰ

367

としては、インクリメント後の値が使用される。ＭＡ１では、@ＳＰ＋，@－ＳＰのモードを指定することはできない。ＭＵＬＸ，ＤＩＶＸの場合、ｓｒｃがＲ０１、ｄｅｓｔがＭ０１に対応し、ｔｍｐはＥＸの中で処理されるものとする。したがって、ｓｒｃ＝@ＳＰ＋の場合に、ｄｅｓｔの実効アドレス計算で参照されるＳＰ値、＊

368

＊およびｔｍｐで参照されるＳＰ値としては、インクリメント後の値が使用される。また、ｔｍｐとｄｅｓｔで同じレジスタを指定した場合には、ｔｍｐの値が消え、最終的にｄｅｓｔの値が残ることになる。

【０５１６】

```
RA1.    src.count.dataの実効アドレス計算==> roladdr
RR1.    @rlladdr==> src1
MA1.    dest. offsetの実効アドレス計算==> moladdr
MR1.    @moladdr==> dest1
EX.     命令固有の動作
            .op.(dest1. src1) ==> dest2
        MULXでは
            .op.(dest1. src1) ==> dest2 . tmp
        DIVXでは
            .op.(dest1. src1. tmp)==>dest2 . tmp
MW1.    dest2 ==> @moladdr
```

ＡＤＤの動作例は図４０６のとおり。このうち、一つのハーフワードの中で、レジスタ指定ＲｇＲ，ＲｇＭと汎用アドレッシング指定ＥａＲ，ＥａＷ，ＳｈＲ，ＳｈＷの両方を含む命令では、ＥａＲ，ＥａＷ，ＳｈＲ，ＳｈＷで@ＳＰ＋，@－ＳＰのモードを指定し、ＲｇＲ，ＲｇＭでＳＰを指定した場合に、ＥａＲ，ＥａＷ，ＳｈＲ，ＳｈＷによるＳＰ値の更新がＲｇＲ，ＲｇＭとしてのＳＰの読みだし値に影響するため、パイプライン実装が難しいという意見がある。具体的に問題となるのは、以下のような命令である。

```
ADD:L   @SP+, SP
SUB:L   @SP+, SP
CMP:L   @SP+, SP
INDEX   #, @SP+, SP
(MOV:Lは含まない)
```

そこで、これらの命令については、動作がインプリメント依存になるものとする。つまり、ＡＤＤ：Ｌ　@ＳＰ＋，ＳＰ実行後のＳＰ値は、インプリメントによって不定の値をとるということにする。ＥＩＴの検出も難しいので、ＥＩＴとはしない。これらの命令は、短縮形が一般形と同じ動作をするという原則には違反することになる。

【０５１７】２オペランドａｄｄｒｅｓｓ～ａｄｄｒｅｓｓ（ＡＡ）

該当命令：

```
QINS    entry. queue
```

ｅｎｔｒｙをＡ０１に、ｑｕｅｕｅをＡ０２に対応させる。キューリンクの実際の操作はＥＸのステップで行なわれる。

```
AA1.    entry の実効アドレス計算==> aoladdr
AA2.    queue の実効アドレス計算==> ao2addr
MA1～MR1.   なし
EX.     命令固有の動作
            .op.(aoladdr, ao2addr)
```

【０５１８】２オペランドａｄｄｒｅｓｓ～ｒｅａｄ（ＡＲ）

該当命令：

```
CSI     comp. update. dest
```

ＣＳＩ命令では、ビットパターンとの関係から、ｄｅｓｔをＡ０１に、ｕｐｄａｔｅをＲ０１に対応させるｃｏｍｐのアクセスと比較、交換の実際の操作は、ＥＸで行なわれる。ＣＳＩ命令のオペランドの処理は、アセンブラ表記とは異なり、ｕｐｄａｔｅの実効アドレス計算、ｄｅｓｔの実効アドレス計算、ｃｏｍｐ値の参照といった順序で行なわれる。ｄｅｓｔでは@ＳＰ＋，@－ＳＰが使用できないが、ｕｐｄａｔｅでは@ＳＰ＋が使用できるので、ｃｏｍｐでＳＰを参照した場合には注意が必要である。

```
RA1.    updateの実効アドレス計算==> roladdr
RR1.    @roladdr==> update1
AA1.    destの実効アドレス計算==> aoladdr
EX.     命令固有の動作
            .op.(aoladdr. update1.comp) ==> comp
```

ＣＳＩの動作例は図４０７のとおり。

【０５１９】１オペランドｒｅａｄ～ｒｅｇｌｉｓｔ（ＲＬ）

該当命令：

```
ENTER   local, reglist
EXITD   reglist. adjsp
```

ＥＮＴＥＲ，ＥＸＩＴＤでは、ｌｏｃａｌ，ａｄｊｓｐをＲ０１に対応させる。ｒｅｇｌｉｓｔの参照とスタックフレームの操作は、ＥＸのステップに含める。したが＊って、ｌｏｃａｌ，ａｄｊｓｐの実効アドレス計算の際に参照されるＳＰ，ＦＰ，Ｒ０～Ｒ１３は、スタックフレーム操作を始めてからの値ではなく、すべて命令実行前の値を使用することになる。ＥＸＩＴＤでは、アセンブラ表記でのオペランド順と逆の順番で実効アドレスが評価されるので、注意が必要である。（ただし、命令再実行の関係で、ｌｏｃａｌ，ａｄｊｓｐはレジスタ直接Ｒｎとイミディエート＃ｉｍｍ＿ｄａｔａのモードしか使用できない。）

```
RA1.    local.adjsp の実効アドレス計算==> roladdr
        （実際にはRnと#imm_dataのモードしか使用できないので、こ
        のステップでは何もアクセスしない。）
RR1.    @roladdr==> src1
EX.     reglist 参照とスタックフレーム操作
          ENTER では
            .op.(src1,R0～R13,FP,SP,reglist)==> FP,SP,stack
          EXITD では
            .op.(src1,FP,SP,stack,reglist)==> R0～R13,FP,SP,PC
```

【０５２０】１オペランドａｄｄｒｅｓｓ～ｒｅｇｌｉｓｔ（ＡＬ）★

該当命令：

```
LDM     src, reglist
STM     reglist,  dest
```

ＬＤＭ，ＳＴＭでは、ｓｒｃ，ｄｅｓｔをＡ０１に対応させる。ｒｅｇｌｉｓｔの参照と実際のレジスタ転送は、ＥＸのステップに含める。したがって、ｓｒｃ，ｄｅｓｔの実効アドレス計算の際に参照されるＳＰ，ＦＰ，Ｒ０～Ｒ１３は、レジスタ転送を始めてからの値ではなく、すべて命令実行前の値を使用することになる。ＬＤＭ　ｒｅｇｌｉｓｔ，@ＳＰ＋とＳＴＭ　ｒｅｇｌｉｓｔ，@－ＳＰについては、ＳＰが複数回更新されるため、一般命令での@ＳＰ＋，@－ＳＰとは異なった扱いをする必要がある。すなわち、モデルの「実効アドレス計算」のステップで@ＳＰ＋，@－ＳＰによるＳＰ値の更新を扱うのではなく、ＥＸのステップでＳＰ値の更新を扱うようにする。このため、ＳＰをＭ０１に対応させる。

〔@ＳＰ＋，@－ＳＰ以外のモードを使用した場合〕

```
AA1.    src.  destの実効アドレス計算==> aoladdr
EX.     reglist 参照とレジスタ転送
          LDM では
            @(aoladdr ～…) ==>REG(reglist)
          STM では
            REG(reglist)==>@(aoladdr～…)
```

【０５２１】〔@ＳＰ＋，@－ＳＰのモードを使用した場合〕

```
MR1.    SP==> tamaddr
EX.     reglist 参照とレジスタ転送
          LDM では
            @ (tmpaddr～…) ==>REG(reglist)
            tmpaddr ＋転送レジスタ数==>tmpaddr
          STM では
            tmpaddr ～転送レジスタ数==>tmpaddr
            REG(reglist)==>@(tmpaddr～…)
```

〔実際のインプリメントでは、レジスタ転送の順序は自由である。これと等価な動作をすればよい。〕

```
MW1.    tmpaddr ==> SP
```

ＬＤＭ　@ＳＰ＋，ｒｅｇｌｉｓｔ，の場合、ｒｅｇｌ

371

ｉｓｔ中にＳＰが指定されていても、最後のMW１のステップでｔｍｐａｄｄｒがＳＰにｏｖｅｒｗｒｉｔｅされるため、結果的にメモリからロードされたＳＰ値が消えてしまうことになる。ＬＤＭ，ＳＴＭの動作例は図４０８のとおり。

【０５２２】付録１３．キャッシュやＴＬＢの整合性確保について

キャッシュやＴＬＢの整合性の確保については、それぞれ関連する命令のところで説明を行なっているが、整理すると以下のようになる。

〔ＡＴを変更した場合の整合性〕

・ＴＬＢ，論理キャッシュ（データキャッシュ）の整合性

－ＬＤＣ，ＬＤＣＴＸ，ＥＩＴ，ＲＥＩＴによりＰＳＷの中のＡＴが変更された場合、ＴＬＢ，論理データキャッシュの整合性が保証される。（ＬＳＩＤがない場合はパージされる。ＬＳＩＤがある場合、ＡＴ＝００の物理空間に対して特別のＬＳＩＤを与えると考えれば、必ずしもパージする必要はない。）

・命令パイプライン、命令キャッシュの整合性

－命令コード整合性の状態は変化しない。ＡＴを変更しても、命令コード整合性が保証されるわけではない。

【０５２３】〔ＵＡＴＢ，ＳＡＴＢの操作をした場合の整合性〕

・ＴＬＢ，論理キャッシュ（データキャッシュ）の整合性

－ＬＤＣ，ＬＤＣＴＸによるＵＡＴＢ，ＳＡＴＢの操作では、ＴＬＢ，論理データキャッシュの整合性が保証される。（ＬＳＩＤがない場合、一般にはパージされる）

・命令パイプライン、命令キャッシュの整合性

－ＬＤＣ，ＬＤＣＴＸによりＵＡＴＢ，ＳＡＴＢを操作しても、命令コード整合性の状態は変化しない。インプリメントによっては、命令キャッシュのパージによって命令コード整合性が良くなる場合もあるが、それが保証されているわけではない。例えば、論理空間Ａ，論理空間Ｂの命令コードをそれぞれ書き換えた後、論理空間ＡでＰＩＢ命令を実行すると、論理空間Ａでの命令コード整合性は保証される。この後ＬＤＣまたはＬＤＣＴＸでＵＡＴＢを操作し、論理空間Ｂに切り換えたとしても、論理空間Ｂでの命令コード整合性は保証されず、それを保証するためには論理空間Ｂでもう一度ＰＩＢ命令を実行する必要がある。ただし、論理空間ＢでＰＩＢ命令を実行してもしなくても、再度ＵＡＴＢを操作して論理空間Ａに戻ってきた場合には、命令コードの整合性が保証されている。実際には、論理命令キャッシュのパージにより、ＵＡＴＢ操作の後は自動的に命令コード整合性が保証されるかもしれないが、プログラミング上はこの機能を当てにしてはいけない。将来ＬＳＩＤを導入してパージを避けることを考えると、一般には、ＵＡＴＢを操作しても命令コード整合性は変わらないと考えなければ

372

ならない。

【０５２４】〔ＡＴＥの操作をした場合の整合性〕

・ＴＬＢ，論理キャッシュ（データキャッシュ）の整合性

－ＬＤＡＴＥによるＡＴＥの操作では、ＴＬＢ，論理データキャッシュの整合性が保証される。（影響のある部分がパージされる）

－ＬＤＡＴＥを用いず、一般のメモリアクセス命令でＡＴＥに使用しているメモリ領域を書き換えた場合には、ＴＬＢ，論理データキャッシュの整合性は保証されない。

・命令パイプライン、命令キャッシュの整合性

－ＡＴＥを更新した場合、そのＡＴＥによりアドレス変換される領域の「命令コードの整合性」は失われる。つまり、その領域のメモリ内容をプログラムとして実行しても、動作は保証されない。これは、ＬＤＡＴＥ命令を用いるかどうかには関係しない。命令コードの整合性を回復する必要があれば、別にＰＩＢ命令を実行する。

【０５２５】〔メモリの操作をした場合の整合性〕

・論理キャッシュ（データキャッシュ）の整合性

－論理アドレスによってメモリをアクセスする場合には、論理データキャッシュの整合性が保証される。（キャッシュの制御機構による）

－ＬＤＰ命令を使って、物理アドレスによりメモリアクセスする場合には、論理データキャッシュの整合性は保証されない。

・命令パイプライン、命令キャッシュの整合性

－メモリ内容を変更した場合、その領域の「命令コードの整合性」は失われる。これは、論理アドレスによるアクセスか物理アドレスによるアクセスかということには関係しない。内容を変更したメモリをプログラムとして実行するには、ＰＩＢ命令を実行し、命令コードの整合性を回復する必要がある。

【０５２６】

【発明の効果】以上のようにこの発明によれば、２つのオペランド間の加算もしくは減算の算術演算を２の補数表現による符号付き二進数として行う命令を指定する第１の命令コードと、２つのオペランドの間の前記算術演算と同一の演算を絶対値表現された符号なし二進数として行う命令を指定する第２の命令コードとが用意された命令セットにより表現されるプログラムを解読して、これら命令コードの各々の指定する算術演算を実行し、各演算によりデスティネーションオペランドに格納された演算結果におけるオーバーフローの発生を示す情報、もしくは各演算の演算結果の正負を示す情報をフラッグで表現したので、加算もしくは減算の算術演算を符号付きで扱いたいか符号なしで扱いたいかは、プログラムにおいて第１の命令コードを用いるか第２の命令コードを用いるかを選択するだけでよく、さらにそれぞれの演算結果のオーバーフローもしくは正負の状態を調べてその後

の処理内容を決定する際にはフラッグを参照するだけでよく、プログラム作成上の負担が少なくなるという効果がある。

【０５２７】また、２つのオペランドの間で２の補数表現による符号付き二進数としてその大小を比較する比較演算を指定する第１の命令コードと、２つのオペランドの間で絶対値表現された符号なし二進数としてその大小を比較する比較演算を指定する第２の命令コードとが用意された命令セットにより表現されるプログラムを解読し、これら命令コードの各々の指定する比較演算を実行し、それぞれ実行された比較演算によるそれぞれの数学的大小関係の情報をフラッグで表現したので、比較演算を符号付きで扱いたいか符号なしで扱いたいかは、プログラムにおいて第１の命令コードを用いるか第２の命令コードを用いるかを選択するだけでよく、さらにそれぞれの数学的大小関係を判断してその後の処理を行う場合でもフラッグを参照するだけでよく、プログラムの作成上の負担が少なくなるという効果がある。

【図面の簡単な説明】

【図１】本発明装置のレジスタセット説明図である。

【図２】本発明装置のレジスタセット説明図である。

【図３】本発明装置のビットについてのデータタイプ説明図である。

【図４】本発明装置のビットフイールドについてのデータタイプ説明図である。

【図５】本発明装置の符号なしビットフィールドについてのデータタイプ説明図である。

【図６】本発明装置の整数についてのデータタイプ説明図である。

【図７】本発明装置の整数についてのデータタイプ説明図である。

【図８】本発明装置の１０進数についてのデータタイプ説明図である。

【図９】本発明装置の１０進数についてのデータタイプ説明図である。

【図１０】本発明装置のストリングについてのデータタイプ説明図である。

【図１１】本発明装置のストリングについてのデータタイプ説明図である。

【図１２】第８図は、本発明装置のキューについてのデータタイプ説明図である。

【図１３】本発明装置の命令フォーマットの記述例を示す説明図である。

【図１４】そのビットパターン図である。

【図１５】本発明装置の命令フォーマット図である。

【図１６】本発明装置の命令フォーマット図である。

【図１７】本発明装置の命令フォーマット図である。

【図１８】本発明装置の命令フォーマット図である。

【図１９】本発明装置の命令フォーマット図である。

【図２０】本発明装置の命令フォーマット図である。

【図２１】本発明装置の命令フォーマット図である。

【図２２】本発明装置の命令フォーマット図である。

【図２３】本発明装置の命令フォーマット図である。

【図２４】本発明装置の命令フォーマット図である。

【図２５】本発明装置の命令フォーマット図である。

【図２６】本発明装置のアドレッシングモードのフォーマット図である。

【図２７】本発明装置のアドレッシングモードのフォーマット図である。

【図２８】本発明装置のアドレッシングモードのフォーマット図である。

【図２９】本発明装置のアドレッシングモードのフォーマット図である。

【図３０】・本発明装置のアドレッシングモードのフォーマット図である。

【図３１】本発明装置のアドレッシングモードのフォーマット図である。

【図３２】本発明装置のアドレッシングモードのフォーマット図である。

【図３３】本発明装置のアドレッシングモードのフォーマット図である。

【図３４】本発明装置のアドレッシングモードのフォーマット図である。

【図３５】本発明装置のアドレッシングモードのフォーマット図である。

【図３６】本発明装置のアドレッシングモードのフォーマット図である。

【図３７】本発明装置のアドレッシングモードのフォーマット図である。

【図３８】本発明装置のローカル変数配置例の説明図である。

【図３９】本発明装置のアドレッシングモードのフォーマット図である。

【図４０】本発明装置のアドレッシングモードのフォーマット図である。

【図４１】本発明装置のアドレッシングモードのフォーマット図である。

【図４２】本発明装置のアドレッシングモードのフォーマット図である。

【図４３】命令ＭＯＶでの注意事項説明図である。

【図４４】ＰＳＷのフォーマット図である。

【図４５】ＰＳＳのフォーマット図である。

【図４６】ＰＳＨのフォーマット図である。

【図４７】命令セットの記述例を示すフォーマット図である。

【図４８】命令ＭＯＶのフォーマット図及びそのフラッグ変化の説明図である。

【図４９】命令ＭＯＶＵのフォーマット図である。

【図５０】そのフラッグ変化の説明図である。

【図５１】命令ＰＵＳＨのフォーマット図である。

375

【図５２】そのフラッグ変化の説明図である。
【図５３】命令ＰＯＰのフォーマット図である。
【図５４】そのフラッグ変化の説明図である。
【図５５】命令ＬＤＭのフォーマット図である。
【図５６】そのフラッグ変化の説明図である。
【図５７】ビットマップ指定の説明図である。
【図５８】命令ＳＴＭのフォーマット図である。
【図５９】そのフラッグ変化の説明図である。
【図６０】ビットマップ指定の説明図である。
【図６１】ビットマップ指定の説明図である。
【図６２】命令ＭＯＶＡのフォーマット図である。
【図６３】そのフラッグ変化の説明図である。
【図６４】命令ＰＵＳＨＡのフォーマット図である。
【図６５】そのフラッグ変化の説明図である。
【図６６】命令ＣＭＰのフォーマット図である。
【図６７】そのフラッグ変化の説明図である。
【図６８】命令ＣＭＰＵのフォーマット図である。
【図６９】そのフラッグ変化の説明図である。
【図７０】命令ＣＨＫのフォーマット図である。
【図７１】そのフラッグ変化の説明図である。
【図７２】命令ＣＨＫのオペレーションの説明図である。
【図７３】命令ＡＤＤのフォーマット図である。
【図７４】そのフラッグ変化の説明図である。
【図７５】命令ＡＤＤＵのフォーマット図である。
【図７６】そのフラッグ変化の説明図である。
【図７７】命令ＡＤＤＸのフォーマット図である。
【図７８】そのフラッグ変化の説明図である。
【図７９】命令ＳＵＢのフォーマット図である。
【図８０】そのフラッグ変化の説明図である。
【図８１】命令ＳＵＢＵのフォーマット図である。
【図８２】そのフラッグ変化の説明図である。
【図８３】命令ＳＵＢＸのフォーマット図である。
【図８４】そのフラッグ変化の説明図である。
【図８５】命令ＭＵＬのフォーマット図である。
【図８６】そのフラッグ変化の説明図である。
【図８７】命令ＭＵＬＵのフォーマット図である。
【図８８】そのフラッグ変化の説明図である。
【図８９】命令ＭＵＬＸのフォーマット図である。
【図９０】そのフラッグ変化の説明図である。
【図９１】命令ＤＩＶのフォーマット図である。
【図９２】そのフラッグ変化の説明図である。
【図９３】命令ＤＩＶＵのフォーマット図である。
【図９４】そのフラッグ変化の説明図である。
【図９５】命令ＤＩＶＸのフォーマット図である。
【図９６】そのフラッグ変化の説明図である。
【図９７】命令ＲＥＭのフォーマット図である。
【図９８】そのフラッグ変化の説明図である。
【図９９】命令ＲＥＭＵのフォーマット図である。
【図１００】そのフラッグ変化の説明図である。

376

【図１０１】命令ＮＥＧのフォーマット図である。
【図１０２】そのフラッグ変化の説明図である。
【図１０３】命令ＩＮＤＥＸのフォーマット図である。
【図１０４】そのフラッグ変化の説明図である。
【図１０５】命令ＡＮＤのフォーマット図である。
【図１０６】そのフラッグ変化の説明図である。
【図１０７】命令ＯＲのフォーマット図である。
【図１０８】そのフラッグ変化の説明図である。
【図１０９】命令ＸＯＲのフォーマット図である。
【図１１０】そのフラッグ変化の説明図である。
【図１１１】命令ＮＯＴのフォーマット図である。
【図１１２】そのフラッグ変化の説明図である。
【図１１３】命令ＳＨＡのフォーマット図である。
【図１１４】そのフラッグ変化の説明図である。
【図１１５】左シフトの説明図である。
【図１１６】右シフトの説明図である。
【図１１７】命令ＳＨＬのフォーマット図である。
【図１１８】そのフラッグ変化の説明図である。
【図１１９】左シフトの説明図である。
【図１２０】右シフトの説明図である。
【図１２１】命令ＲＯＴのフォーマット図である。
【図１２２】そのフラッグ変化の説明図である。
【図１２３】左回転の説明図である。
【図１２４】右回転の説明図である。
【図１２５】命令ＳＨＸＬのフォーマット図である。
【図１２６】そのフラッグ変化の説明図である。
【図１２７】命令ＸＨＸＬのフォーマット図である。
【図１２８】命令ＸＨＸＲのフォーマット図である。
【図１２９】そのフラッグ変化の説明図である。
【図１３０】命令ＳＨＸＲのフォーマット図である。
【図１３１】命令ＲＶＢＹのフォーマット図である。
【図１３２】そのフラッグ変化の説明図である。
【図１３３】命令ＲＶＢＩのフォーマット図である。
【図１３４】そのフラッグ変化の説明図である。
【図１３５】ビット操作命令の説明図である。
【図１３６】ビット操作命令の説明図である。
【図１３７】命令ＢＴＳＴのフォーマット図である。
【図１３８】そのフラッグ変化の説明図である。
【図１３９】命令ＢＳＥＴのフォーマット図である。
【図１４０】そのフラッグ変化の説明図である。
【図１４１】命令ＢＣＬＲのフォーマット図である。
【図１４２】そのフラッグ変化の説明図である。
【図１４３】命令ＢＮＯＴのフォーマット図である。
【図１４４】そのフラッグ変化の説明図である。
【図１４５】命令ＢＳＣＨのフォーマット図である。
【図１４６】そのフラッグ変化の説明図である。
【図１４７】固定長ビットフィールド操作命令の説明図である。
【図１４８】ビットフィールド命令のフォーマット図である。

377

【図１４９】ビットフィールド命令のフォーマット図である。
【図１５０】命令ＢＦＥＸＴのフォーマット図である。
【図１５１】そのフラッグ変化の説明図である。
【図１５２】命令ＢＦＥＸＴＵのフォーマット図である。
【図１５３】そのフラッグ変化の説明図である。
【図１５４】命令ＢＦＩＮＳのフォーマット図である。
【図１５５】そのフラッグ変化の説明図である。
【図１５６】命令ＢＦＩＮＳＵのフォーマット図である。
【図１５７】そのフラッグ変化の説明図である。
【図１５８】命令ＢＦＣＭＰのフォーマット図である。
【図１５９】そのフラッグ変化の説明図である。
【図１６０】命令ＢＦＣＭＰＵのフォーマット図である。
【図１６１】そのフラッグ変化の説明図である。
【図１６２】命令ＢＶＳＣＨのフォーマット図である。
【図１６３】そのフラッグ変化の説明図である。
【図１６４】命令ＢＶＭＡＰのフォーマット図である。
【図１６５】そのフラッグ変化の説明図である。
【図１６６】命令ＢＶＭＡＰのフォーマット図である。
【図１６７】命令ＢＶＭＡＰのフォーマット図である。
【図１６８】命令ＢＶＭＡＰのフォーマット図である。
【図１６９】命令ＢＶＣＰＹのフォーマット図である。
【図１７０】そのフラッグ変化の説明図である。
【図１７１】命令ＢＶＰＡＴのフォーマット図である。
【図１７２】そのフラッグ変化の説明図である。
【図１７３】命令ＡＤＤＤＸのフォーマット図である。
【図１７４】そのフラッグ変化の説明図である。
【図１７５】命令ＳＵＢＤＸのフォーマット図である。
【図１７６】そのフラッグ変化の説明図である。
【図１７７】命令ＰＡＣＫｓｓのフォーマット図である。
【図１７８】そのフラッグ変化の説明図である。
【図１７９】命令ＵＮＰＫｓｓのフォーマット図である。
【図１８０】そのフラッグ変化の説明図である。
【図１８１】命令ＵＮＰＫｓｓの説明図である。
【図１８２】命令ＳＭＯＶのフォーマット図である。
【図１８３】そのフラッグ変化の説明図である。
【図１８４】命令ＳＣＭＰの説明図である。
【図１８５】そのフラッグ変化の説明図である。
【図１８６】そのフラッグ変化の説明図である。
【図１８７】命令ＳＳＣＨのフォーマット図である。
【図１８８】そのフラッグ変化の説明図である。
【図１８９】命令ＳＳＴＲのフォーマット図である。
【図１９０】そのフラッグ変化の説明図である。
【図１９１】命令ＱＩＮＳのフォーマット図である。
【図１９２】そのフラッグ変化の説明図である。

378

【図１９３】命令ＱＩＮＳの説明図である。
【図１９４】命令ＱＩＮＳの説明図である。
【図１９５】命令ＱＩＮＳの説明図である。
【図１９６】命令ＱＤＥＬのフォーマット図である。
【図１９７】そのフラッグ変化の説明図である。
【図１９８】命令ＱＤＥＬの説明図である。
【図１９９】命令ＱＤＥＬの説明図である。
【図２００】命令ＱＤＥＬの説明図である。
【図２０１】命令ＱＳＣＨのフォーマット図である。
【図２０２】そのフラッグ変化の説明図である。
【図２０３】命令ＱＳＣＨの説明図である。
【図２０４】命令ＱＳＣＨの説明図である。
【図２０５】命令ＱＳＣＨの説明図である。
【図２０６】命令ＢＲＡのフォーマット図である。
【図２０７】そのフラッグ変化の説明図である。
【図２０８】命令Ｂｃｃのフォーマット図である。
【図２０９】そのフラッグ変化の説明図である。
【図２１０】命令ＢＳＲのフォーマット図である。
【図２１１】そのフラッグ変化の説明図である。
【図２１２】命令ＪＭＰのフォーマット図である。
【図２１３】そのフラッグ変化の説明図である。
【図２１４】命令ＪＳＲのフォーマット図である。
【図２１５】そのフラッグ変化の説明図である。
【図２１６】命令ＡＣＢのフォーマット図である。
【図２１７】そのフラッグ変化の説明図である。
【図２１８】命令ＡＣＢの説明図である。
【図２１９】命令ＳＣＢのフォーマット図である。
【図２２０】そのフラッグ変化の説明図である。
【図２２１】命令ＥＮＴＥＲのフォーマット図である。
【図２２２】そのフラッグ変化の説明図である。
【図２２３】命令ＥＮＴＥＲの説明図である。
【図２２４】命令ＥＸＩＴＤの説明図である。
【図２２５】そのフラッグ変化の説明図である。
【図２２６】命令ＥＸＩＴＤの説明図である。
【図２２７】命令ＲＴＳのフォーマット図である。
【図２２８】そのフラッグ変化の説明図である。
【図２２９】命令ＮＯＰのフォーマット図である。
【図２３０】そのフラッグ変化の説明図である。
【図２３１】命令ＰＩＢのフォーマット図である。
【図２３２】そのフラッグ変化の説明図である。
【図２３３】命令ＢＳＥＴＩのフォーマット図である。
【図２３４】そのフラッグ変化の説明図である。
【図２３５】命令ＢＣＬＲＩのフォーマット図である。
【図２３６】そのフラッグ変化の説明図である。
【図２３７】命令ＣＳＩのフォーマット図である。
【図２３８】そのフラッグ変化の説明図である。
【図２３９】命令ＬＤＣのフォーマット図である。
【図２４０】そのフラッグ変化の説明図である。
【図２４１】命令ＳＴＣのフォーマット図である。
【図２４２】そのフラッグ変化の説明図である。

379

【図２４３】命令ＬＤＰＳＢのフォーマット図である。
【図２４４】そのフラッグ変化の説明図である。
【図２４５】命令ＬＤＰＳＭのフォーマット図である。
【図２４６】そのフラッグ変化の説明図である。
【図２４７】命令ＳＴＰＳＢのフォーマット図である。
【図２４８】そのフラッグ変化の説明図である。
【図２４９】命令ＳＴＰＳＭのフォーマット図である。
【図２５０】そのフラッグ変化の説明図である。
【図２５１】命令ＬＤＰのフォーマット図である。
【図２５２】そのフラッグ変化の説明図である。
【図２５３】命令ＳＴＰのフォーマット図である。
【図２５４】そのフラッグ変化の説明図である。
【図２５５】命令ＪＲＮＧのフォーマット図である。
【図２５６】そのフラッグ変化の説明図である。
【図２５７】命令ＪＲＮＧのフォーマット図である。
【図２５８】命令ＪＲＮＧのフォーマット図である。
【図２５９】命令ＪＲＮＧのフォーマット図である。
【図２６０】命令ＪＲＮＧのフォーマット図である。
【図２６１】命令ＪＲＮＧのフォーマット図である。
【図２６２】命令ＪＲＮＧのフォーマット図である。
【図２６３】命令ＲＲＮＧのフォーマット図である。
【図２６４】そのフラッグ変化の説明図である。
【図２６５】命令ＲＲＮＧの説明図である。
【図２６６】命令ＲＲＮＧの説明図である。
【図２６７】命令ＲＲＮＧの説明図である。
【図２６８】命令ＴＲＡＰＡのフォーマット図である。
【図２６９】そのフラッグ変化の説明図である。
【図２７０】命令ＴＲＡＰのフォーマット図である。
【図２７１】そのフラッグ変化の説明図である。
【図２７２】命令ＲＥＩＴのフォーマット図である。
【図２７３】そのフラッグ変化の説明図である。
【図２７４】命令ＲＥＩＴのフォーマット図である。
【図２７５】命令ＷＡＩＴのフォーマット図である。
【図２７６】そのフラッグ変化の説明図である。
【図２７７】命令ＬＤＣＴＸのフォーマット図である。
【図２７８】そのフラッグ変化の説明図である。
【図２７９】命令ＳＴＣＴＸのフォーマット図である。
【図２８０】そのフラッグ変化の説明図である。
【図２８１】命令ＡＣＳのフォーマット図である。
【図２８２】そのフラッグ変化の説明図である。
【図２８３】命令ＭＯＶＰＡのフォーマット図である。
【図２８４】そのフラッグ変化の説明図である。
【図２８５】命令ＭＯＶＰＡのフォーマット図である。
【図２８６】命令ＭＯＶＰＡのフォーマット図である。
【図２８７】命令ＬＤＡＴＥの説明図である。
【図２８８】そのフラッグ変化の説明図である。
【図２８９】そのフラッグ変化の説明図である。
【図２９０】命令ＳＴＡＴＥのフォーマット図である。
【図２９１】そのフラッグ変化の説明図である。
【図２９２】そのフラッグ変化の説明図である。

380

【図２９３】命令ＰＴＬＢのフォーマット図である。
【図２９４】そのフラッグ変化の説明図である。
【図２９５】命令ＰＳＴＬＢのフォーマット図である。
【図２９６】そのフラッグ変化の説明図である。
【図２９７】ＡＴフィールドの説明図である。
【図２９８】論理アドレスの説明図である。
【図２９９】ページアドレスの説明図である。
【図３００】多重論理空間の説明図である。
【図３０１】ＵＡＴＢのフォーマット図である。
【図３０２】ＳＡＴＢのフォーマット図である。
【図３０３】ＳＸの制限を示す説明図である。
【図３０４】テーブル領域の説明図である。
【図３０５】テーブル領域の説明図である。
【図３０６】テーブル領域の説明図である。
【図３０７】ＳＴＥのフォーマット図である。
【図３０８】ＰＸの制限を示す説明図である。
【図３０９】ＳＴＥの説明図である。
【図３１０】論理アドレスの説明図である。
【図３１１】ＰＴＥのフォーマット図である。
【図３１２】ＰＴＥの各値とアクセス可能リングとの関係を示す説明図である。
【図３１３】ＰＴＥの説明図である。
【図３１４】本発明装置の論理アドレス拡張に係るメモリマップの図である。
【図３１５】本発明装置の論理アドレス拡張に係るメモリマップの図である。
【図３１６】データ転送命令のフラッグ変化の説明図である。
【図３１７】比較テスト命令のフラッグ変化の説明図である。
【図３１８】算術演算命令のフラッグ変化の説明図である。
【図３１９】論理演算命令のフラッグ変化の説明図である。
【図３２０】シフト命令のフラッグ変化の説明図である。
【図３２１】ビット操作命令のフラッグ変化の説明図である。
【図３２２】固定長ビットフィールド命令のフラッグ変化の説明図である。
【図３２３】固定長ビットフィールド命令のフラッグ変化の説明図である。
【図３２４】任意長ビットフィールド命令のフラッグ変化の説明図である。
【図３２５】１０進演算命令のフラッグ変化の説明図である。
【図３２６】ストリング命令のフラッグ変化の説明図である。
【図３２７】キュー操作命令のフラッグ変化の説明図である。

381

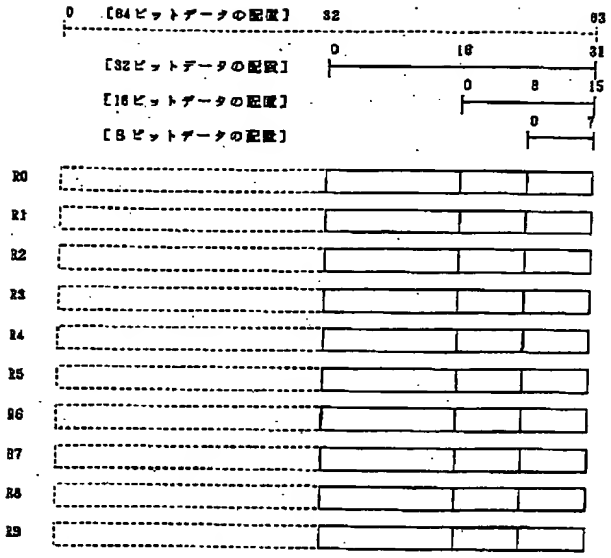
【図３２８】ジャンプ命令のフラッグ変化の説明図である。
【図３２９】マルチプロセッサ命令のフラッグ変化の説明図である。
【図３３０】制御空間、物理空間操作命令のフラッグ変化の説明図である。
【図３３１】ＯＳ関連命令のフラッグ変化の説明図である。
【図３３２】MMU関連命令のフラッグ変化の説明図である。
【図３３３】サブルーチンコールの説明図である。
【図３３４】スタックフレームの説明図である。
【図３３５】命令シーケンスの説明図である。
【図３３６】命令シーケンスの説明図である。
【図３３７】プログラム例を示す説明図である。
【図３３８】サブルーチンコールの説明図である。
【図３３９】制御空間の説明図である。
【図３４０】ＰＳＷのフォーマット図である。
【図３４１】ＩＭＡＳＫのフォーマット図である。
【図３４２】ＳＭＲＮＧのフォーマット図である。
【図３４３】ＣＴＸＢＢのフォーマット図である。
【図３４４】ＤＩのフォーマット図である。
【図３４５】ＣＳＷのフォーマット図である。
【図３４６】ＤＣＥのフォーマット図である。
【図３４７】ＣＴＸＢＦＭのフォーマット図である。
【図３４８】ＥＩＴＶＢのフォーマット図である。
【図３４９】ＪＲＮＧＶＢのフォーマット図である。
【図３５０】ＳＰ０～ＳＰ３のフォーマット図である。
【図３５１】ＳＰＩのフォーマット図である。
【図３５２】ＩＯＡＤＤＲ，ＩＯＭＡＳＫのフォーマット図である。
【図３５３】ＵＡＴＢのフォーマット図である。
【図３５４】ＳＡＴＢのフォーマット図である。
【図３５５】ＬＳＩＤのフォーマット図である。
【図３５６】ＣＴＸＢのフォーマット図である。
【図３５７】ＣＴＸＢＦＭのフォーマット図である。
【図３５８】ＥＩＴＶＴＥのフォーマット図である。
【図３５９】スタックフレームの説明図である。
【図３６０】ＥＩＴのスタックフォーマット図である。
【図３６１】ＥＩＴのスタックフォーマット図である。
【図３６２】ＩＯＩＮＦのフォーマット図である。
【図３６３】ＥＩＴのベクトルテーブルの図である。
【図３６４】ＥＩＴのベクトルテーブルの図である。
【図３６５】ＪＲＮＧの説明図である。
【図３６６】ＥＩＴの説明図である。
【図３６７】ＥＩＴの説明図である。
【図３６８】ＩＭＡＳＫの説明図である。
【図３６９】システムコールの説明図である。
【図３７０】システムコールの説明図である。
【図３７１】ＤＣＥの説明図である。

382

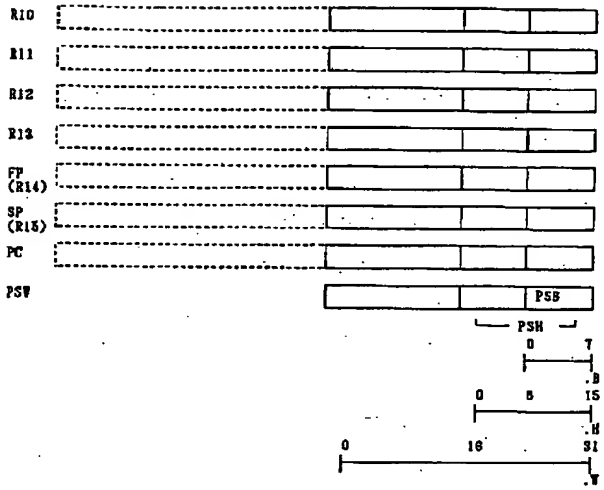
【図３７２】ＤＣＥ，ＤＩ，ＥＩの比較図である。
【図３７３】ＤＣＥの使用例の説明図である。
【図３７４】ビット割当図である。
【図３７５】ビット割当図である。
【図３７６】ビット割当図である。
【図３７７】ビット割当図である。
【図３７８】ビット割当図である。
【図３７９】ビット割当図である。
【図３８０】ビット割当図である。
【図３８１】ビット割当図である。
【図３８２】ビット割当図である。
【図３８３】ビット割当図である。
【図３８４】ビット割当図である。
【図３８５】ビット割当図である。
【図３８６】オペランドフィールド名索引図である。
【図３８７】オペランドフィールド名索引図である。
【図３８８】オペランドフィールド名索引図である。
【図３８９】ｃｃｃｃの割当て図である。
【図３９０】ｅｅｅｅの割当て図である。
【図３９１】Ｍ＿ｆｌａｇの説明図である。
【図３９２】ＢＶＭＡＰ命令の演算コード図である。
【図３９３】アドレッシングモードの対応図である。
【図３９４】アドレッシングモードの対応図である。
【図３９５】アドレッシングモードの対応図である。
【図３９６】命令実行モデルのオペランドとの対応図である。
【図３９７】ＰＵＳＨＡ　＠　ＳＰ　等の説明図である。
【図３９８】ＰＯＰ命令の説明図である。
【図３９９】ＭＯＶ命令の説明図である。
【図４００】ＰＡＣＫ命令等の説明図である。
【図４０１】ＳＴＣ命令等の説明図である。
【図４０２】ＢＦＩＮＳ命令等の説明図である。
【図４０３】ＣＭＰ命令の説明図である。
【図４０４】ＩＮＤＥＸ命令の説明図である。
【図４０５】ＡＣＢ命令等の説明図である。
【図４０６】ＡＤＤ命令の説明図である。
【図４０７】ＣＳＩ命令の説明図である。
【図４０８】ＬＤＭ命令等の説明図である。
【図４０９】四則演算命令の動作説明図である。
【図４１０】四則演算命令の動作説明図である。
【図４１１】四則演算命令の動作説明図である。
【図４１２】四則演算命令の動作説明図である。
【図４１３】四則演算命令の動作説明図である。
【図４１４】四則演算命令の動作説明図である。
【図４１５】オーバーフロー時のフラッグ説明図である。
【図４１６】オーバーフロー時のフラッグ説明図である。
【図４１７】オーバーフロー時のフラッグ説明図であ

る。

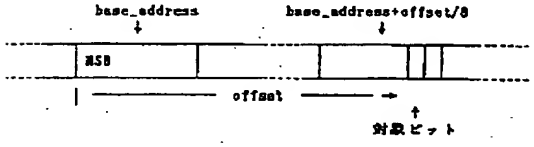
【図１】



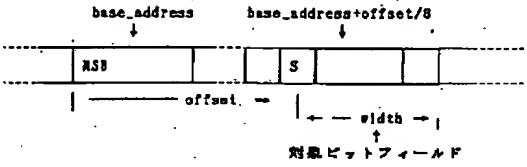
【図２】



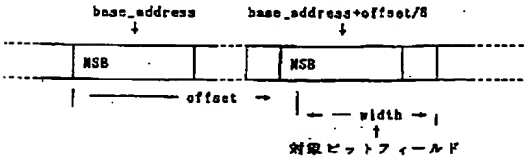
【図３】



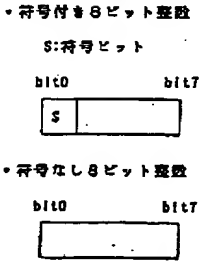
【図４】



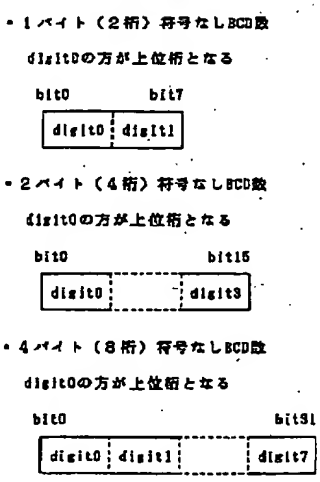
【図５】



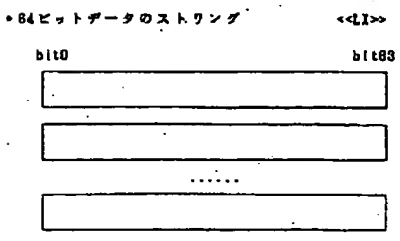
【図６】



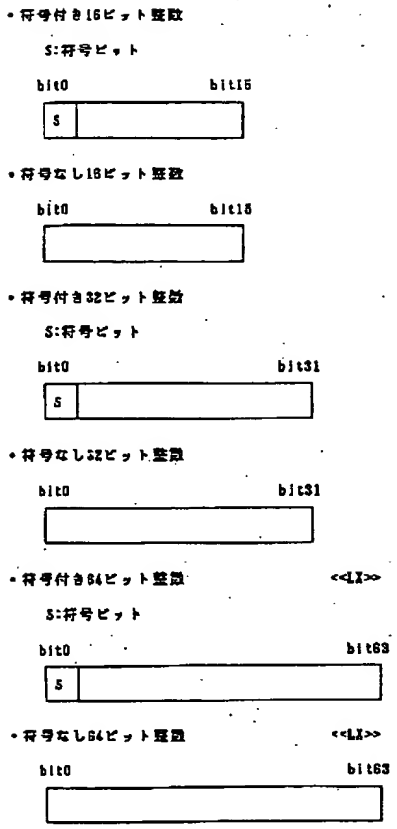
【図８】



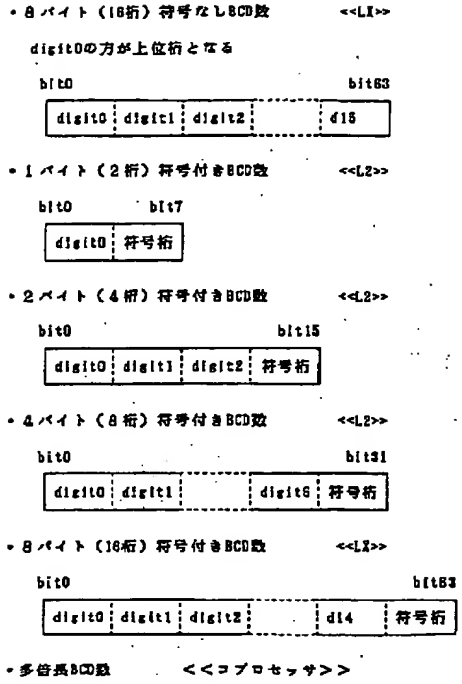
【図１１】



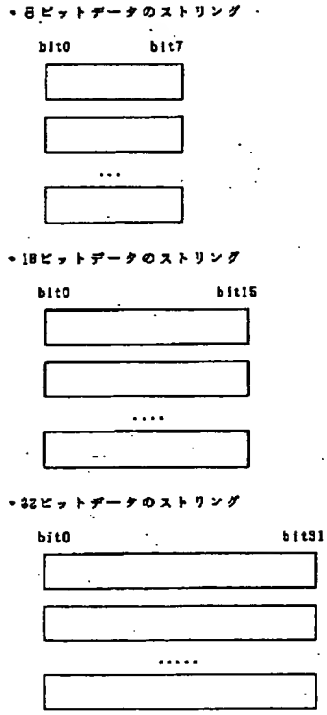
【図７】



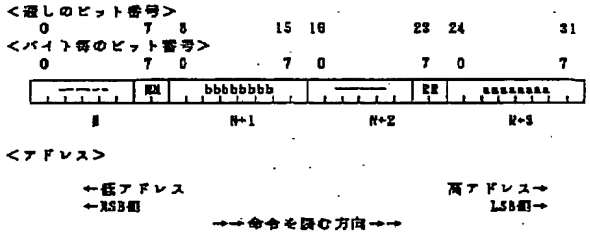
【図９】



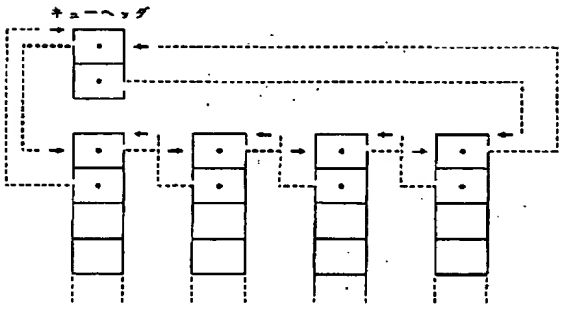
【図１０】



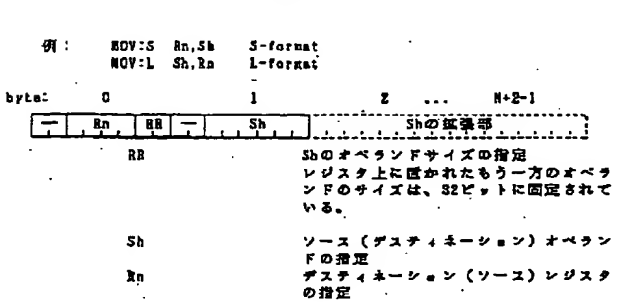
【図１３】



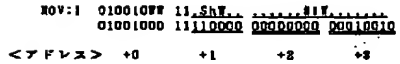
【図１２】



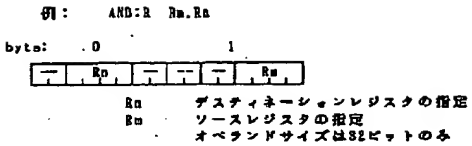
【図１５】



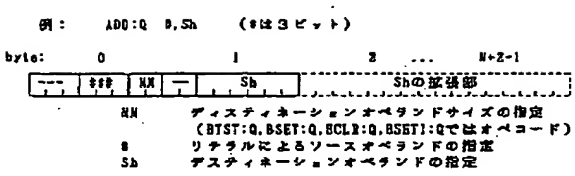
【図１４】



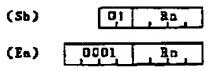
【図１６】



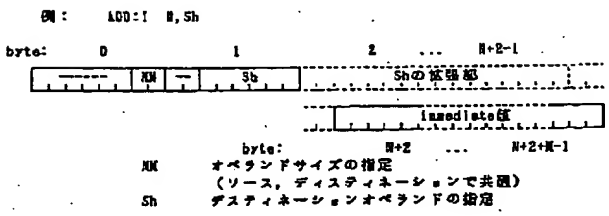
【図１７】



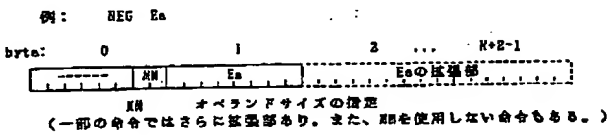
【図２６】



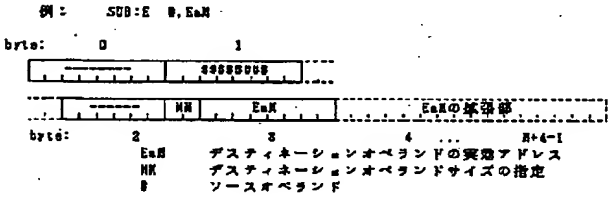
【図１８】



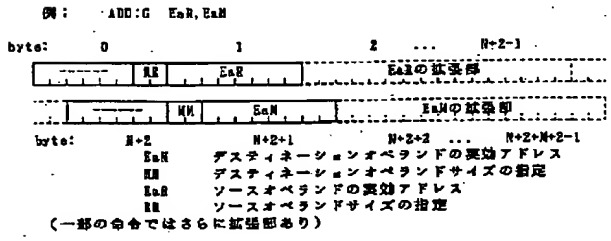
【図１９】



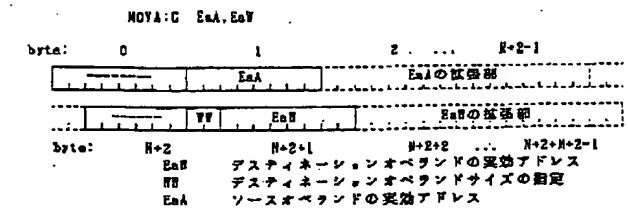
【図２１】



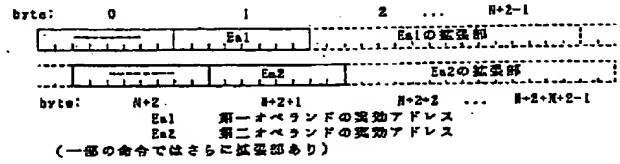
【図２０】



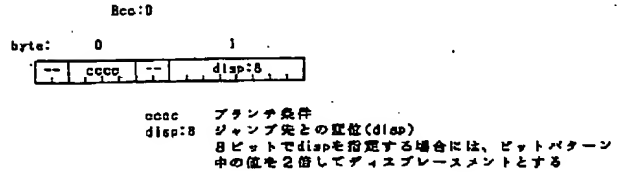
【図２２】



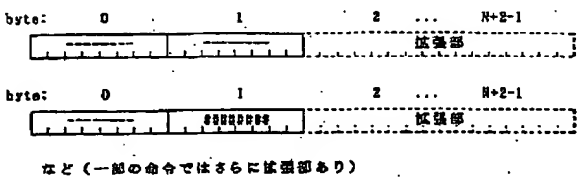
【図２３】



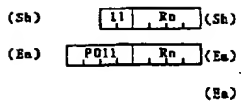
【図２４】



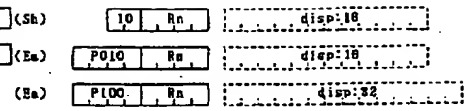
【図２５】



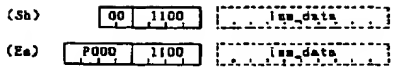
【図２７】



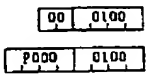
【図２８】



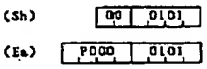
【図２９】



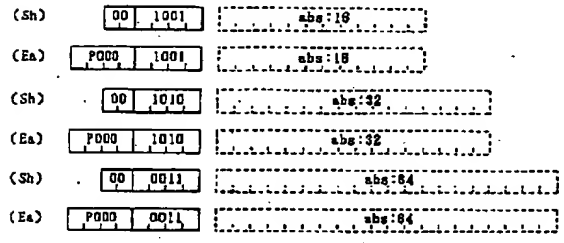
【図３２】



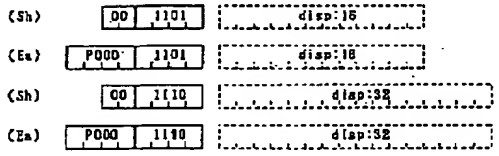
【図３３】



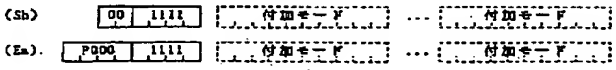
【図３０】



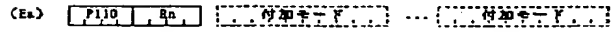
【図３１】



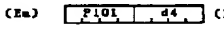
【図３５】



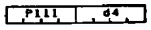
【図３４】



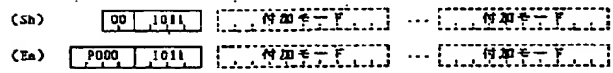
【図３７】



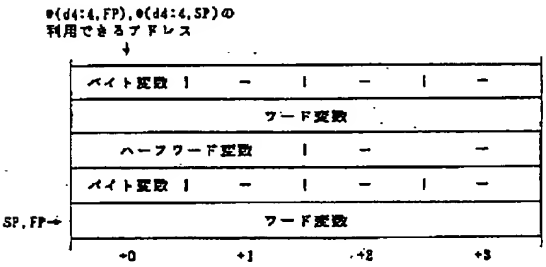
【図３９】



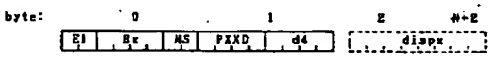
【図３６】



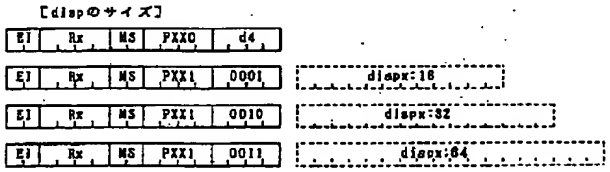
【図３８】



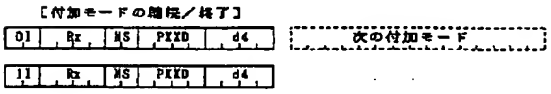
【図４０】



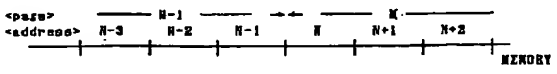
【図４２】



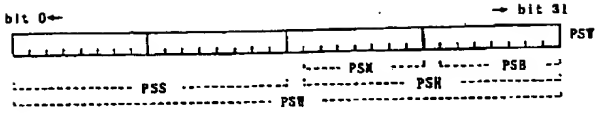
【図４１】



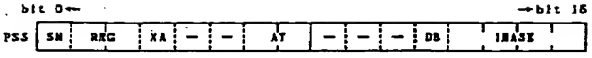
【図４３】



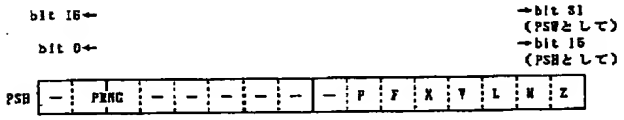
【図４４】



【図４５】

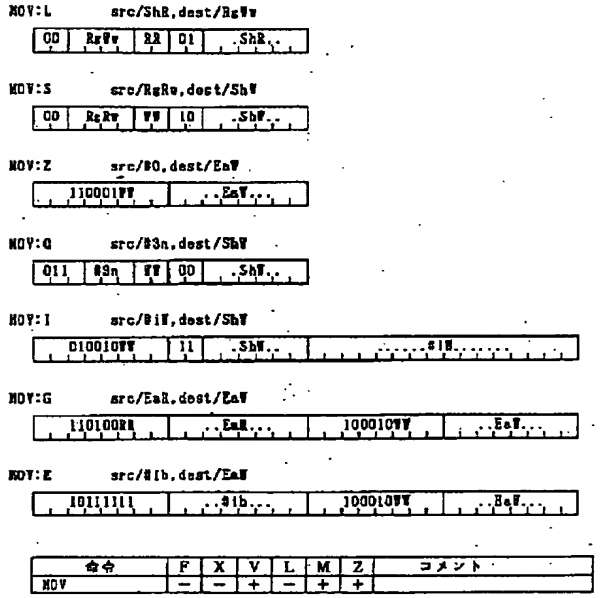


【図４６】

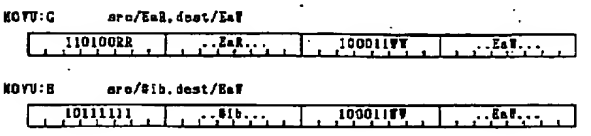


【図４７】

【図４８】

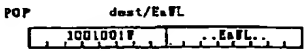


【図４９】

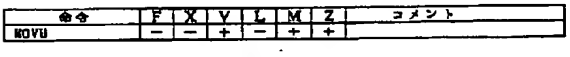


【図５２】

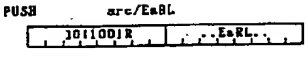
【図５３】



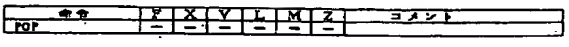
【図５０】



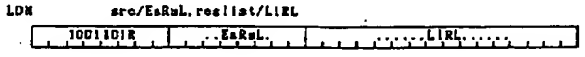
【図５１】



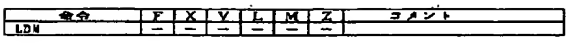
【図５４】



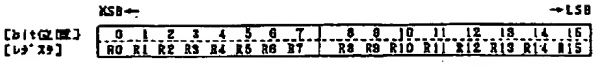
【図５５】



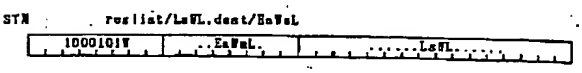
【図５６】



【図５７】

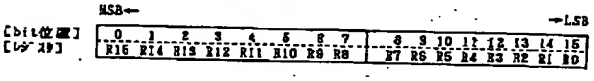


【図５８】

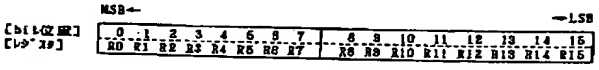


【図５９】

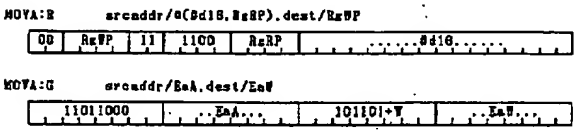
【図６０】



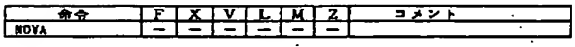
【図６１】



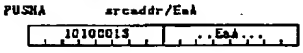
【図６２】



【図６３】

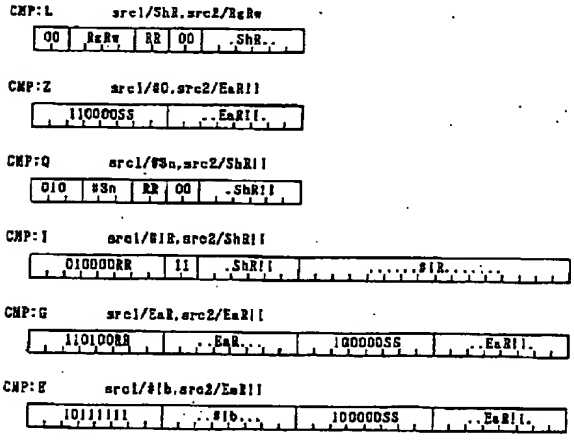


【図６４】

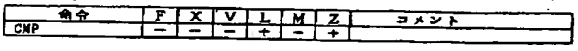


【図６５】

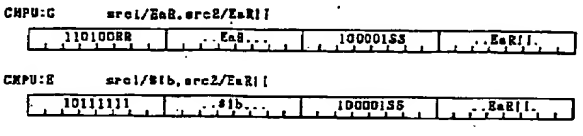
【図６６】



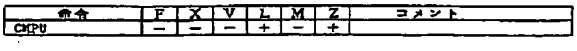
【図６７】



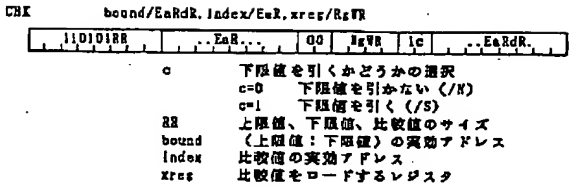
【図６８】



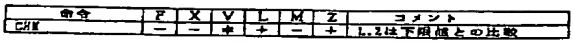
【図６９】



【図７０】



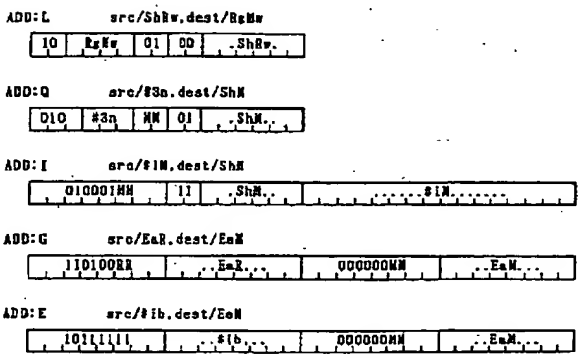
【図７１】



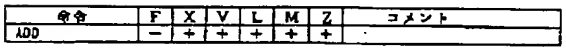
【図７２】

	Index＜下限値	下限値≦Index Index＜上限値	上限値≦Index
L_flag	1	0	0①
V_flag	1	0	1

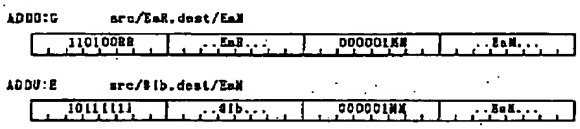
【図７３】



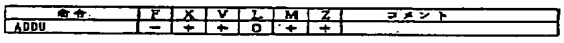
【図７４】



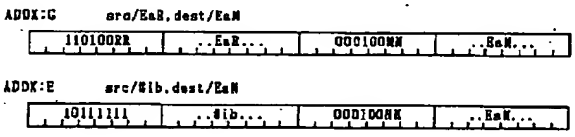
【図７５】



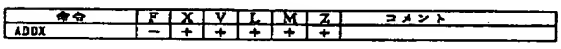
【図７６】



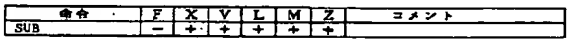
【図７７】



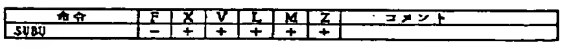
【図７８】



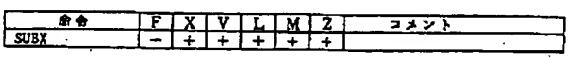
【図８０】



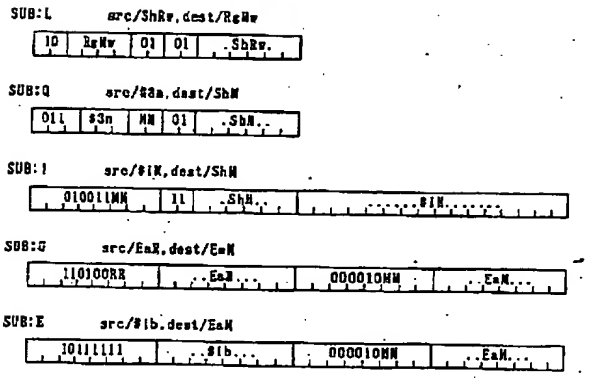
【図８２】



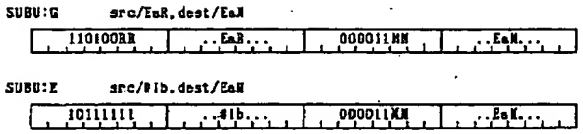
【図８４】



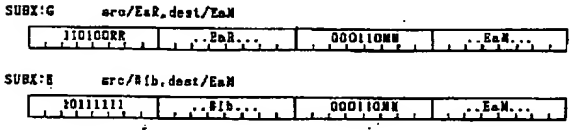
【図７９】



【図８１】



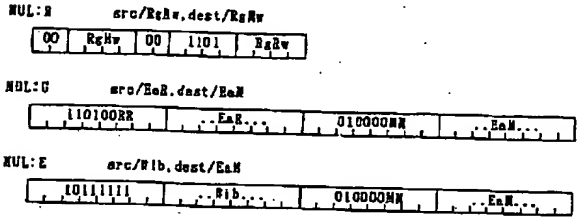
【図８３】



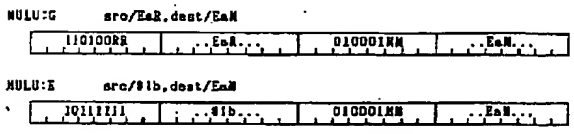
【図８６】

命令	F	X	V	L	M	Z	コメント
MUL	−	−	+	÷	+	+	

【図８５】



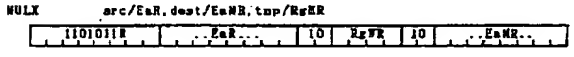
【図８７】



【図８８】

命令	F	X	V	L	M	Z	コメント
MULU	−	−	+	0	+	+	

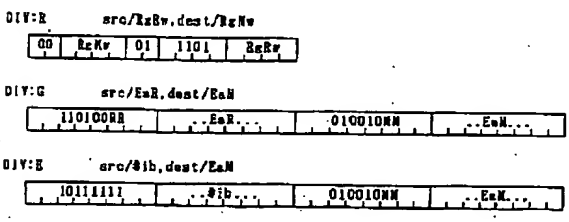
【図８９】



【図９０】

命令	F	X	V	L	M	Z	コメント
MULX	*	−	0	0	+	+	H,Zはdest基準、Fはtmp=0

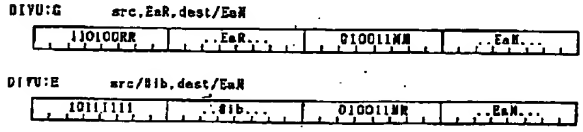
【図９１】



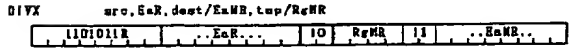
【図９２】

命令	F	X	V	L	M	Z	コメント
DIV	−	−	0	+	+	+	
	−	−	1	0	1	0	-(最小負数)÷(-1)の場合
	−	−	1	−	−	−	-0除算，EITの場合

【図９３】



【図９５】



【図９４】

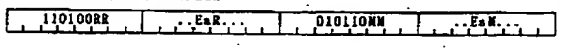
命令	F	X	V	L	M	Z	コメント
DIVU	－	－	0	0	＋	＋	－0除算，EITの場合
	－	－	1	－	－	－	

【図９６】

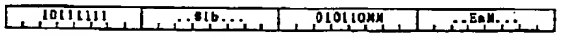
命令	F	X	V	L	M	Z	コメント
DIVX	＊	－	0	0	＋	＋	N,Zはdest基準、Fはtmp=0
	－	－	1	－	－	－	-destが"-79-"の場合
	－	－	1	－	－	－	－0除算，EITの場合

【図９７】

REM:G　　src/EaR,dest/EaN



REM:E　　src/#ib,dest/EaN

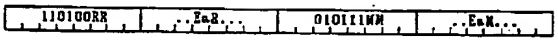


【図９８】

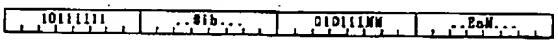
命令	F	X	V	L	M	Z	コメント
REM	－	－	0	＋	＋	＋	
	－	－	0	－	－	－	－0除算，EITの場合

【図９９】

REMU:G　　src/EaR,dest/EaN



REMU:E　　src/#ib,dest/EaN

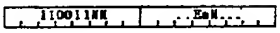


【図１００】

命令	F	X	V	L	M	Z	コメント
REMU	－	－	0	0	＋	＋	
	－	－	0	－	－	－	－0除算，EITの場合

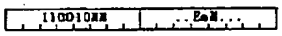
【図１１１】

NOT　　dest/EaN



【図１０１】

NEG　　dest/EaN

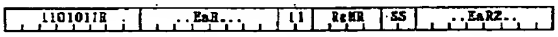


【図１０２】

命令	F	X	V	L	M	Z	コメント
NEG	－	－	＋	＋	＋	＋	

【図１０３】

INDEX　　indexsize/EaR,subscript/EaRZ,xreg/RgNN



R　xregとindexsizeのサイズ
　R=0 32ビット
　R=1 64ビット　<<LX>>
SS　subscriptのサイズ
xreg アドレス計算用アキュムレータ

【図１０４】

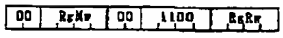
命令	F	X	V	L	M	Z	コメント
INDEX	－	－	＋	＋	＋	＋	N,Zはxreg基準

【図１０６】

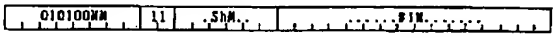
命令	F	X	V	L	M	Z	コメント
AND	－	－	－	－	＋	＋	

【図１０５】

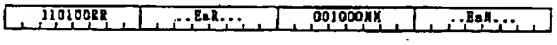
AND:B　　src/RgRw,dest/RgRw



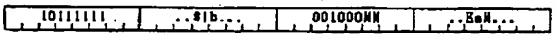
AND:I　　src/#IN,dest/ShN



AND:G　　src/EaR,dest/EaN

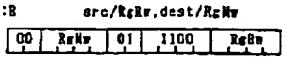


AND:E　　src/#ib,dest/EaN

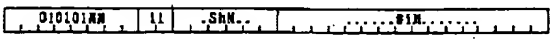


【図１０７】

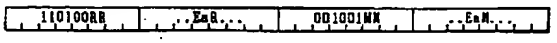
OR:B　　src/RgRw,dest/RgRw



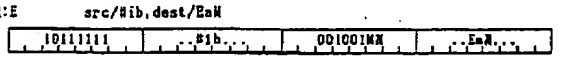
OR:I　　src/#IN,dest/ShN



OR:G　　src/EaR,dest/EaN

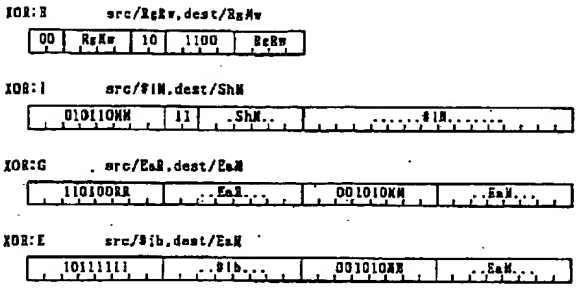


OR:E　　src/#ib,dest/EaN



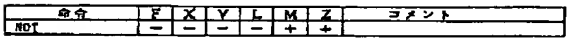
【図１０８】

【図１０９】

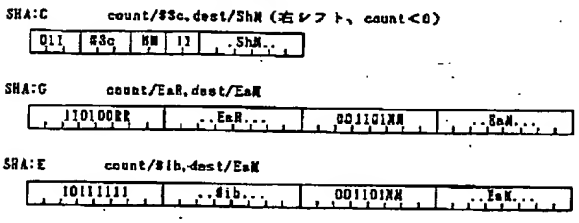


【図１１０】

【図１１２】

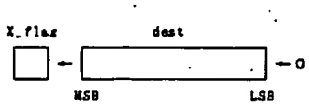


【図１１３】

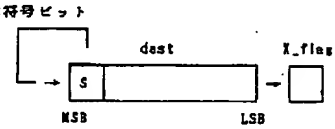


【図１１４】

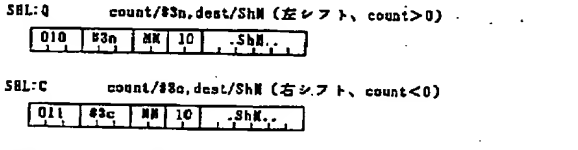
【図１１５】



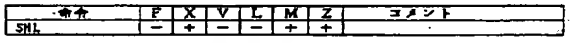
【図１１６】



【図１１７】



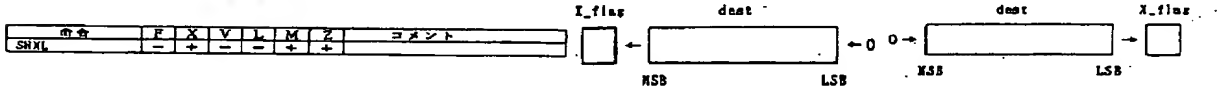
【図１１８】



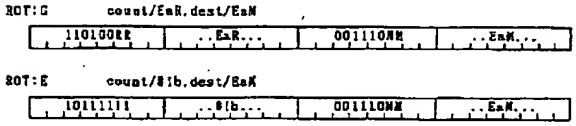
【図１１９】

【図１２０】

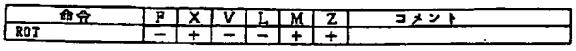
【図１２６】



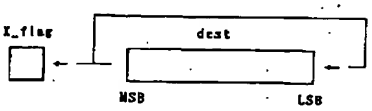
【図１２１】



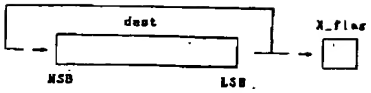
【図１２２】



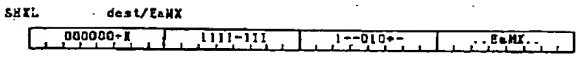
【図１２３】



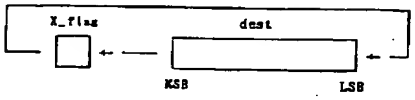
【図１２４】



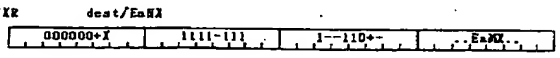
【図１２５】



【図１２７】



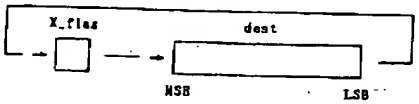
【図１２８】



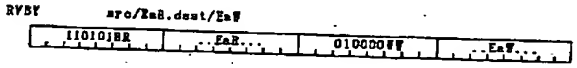
【図１２９】

命令	F	X	V	L	M	Z	コメント
SHXR	−	+	−	−	+	+	

【図１３０】



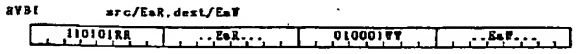
【図１３１】



【図１３２】

命令	F	X	V	L	M	Z	コメント
RVBY	−	−	−	−	−	−	

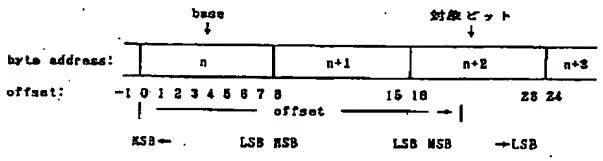
【図１３３】



【図１３４】

命令	F	X	V	L	M	Z	コメント
RVBI	−	−	−	−	−	−	

【図１３５】

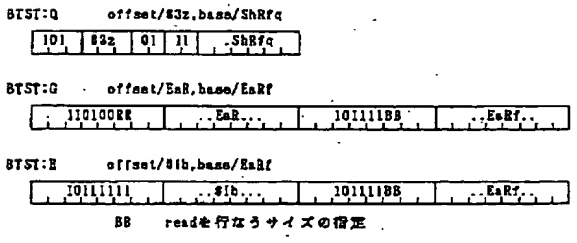


【図１３６】

	レジスタ対象	メモリ対象
アクセスサイズ.B	OK	OK
アクセスサイズ.H	OK	OK<<L2>>
アクセスサイズ.W	OK	OK<<L2>>
アクセスサイズ.L	<<LX>>	<<LX>>

アセンブラでのデフォルトはすべて.B

【図１３７】



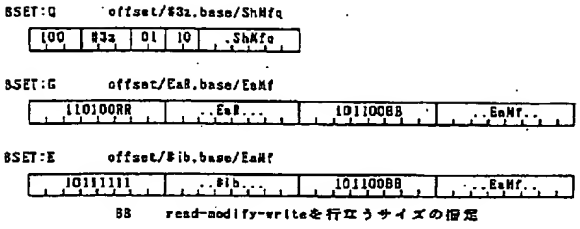
【図１３８】

命令	F	X	V	L	M	Z	コメント
BTST	−	−	−	−	−	+	Zがテスト結果

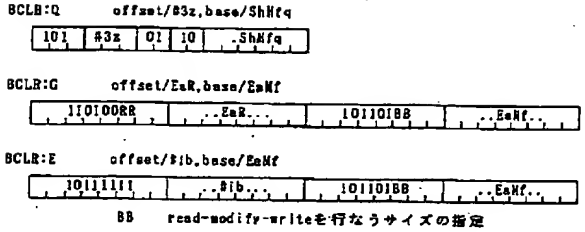
【図１４０】

命令	F	X	V	L	M	Z	コメント
BSET	−	−	−	−	−	+	Zがテスト結果

【図１３９】



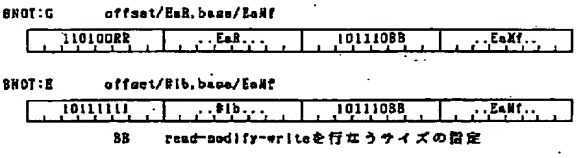
【図１４１】



【図１４２】

命令	F	X	V	L	M	Z	コメント
BCLR	−	−	−	−	−	+	Zはテスト結果

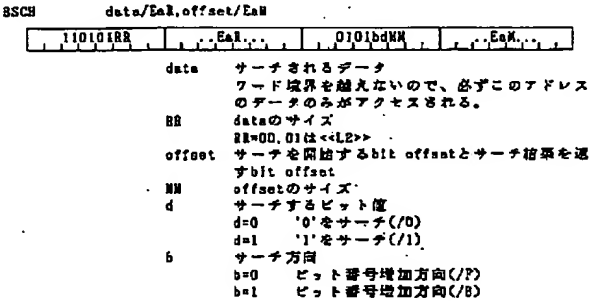
【図１４３】



【図１４４】

命令	F	X	V	L	M	Z	コメント
BNOT	−	−	−	−	−	+	Zはテスト結果

【図１４５】



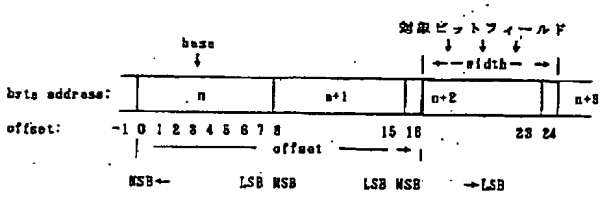
【図１４６】

命令	F	X	V	L	M	Z	コメント
BSCH	−	−	*	−	−	−	Vはサーチ失敗

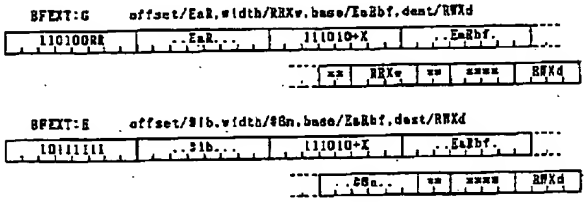
【図１５１】

命令	F	X	V	L	M	Z	コメント
BFEXT	−	−	+	−	+	+	

【図１４７】



【図１５０】



【図１５３】

命令	F	X	V	L	M	Z	コメント
BFEXTU	−	−	+	−	+	+	

【図１５５】

命令	F	X	V	L	M	Z	コメント
BFINS	−	−	+	−	+	+	

【図１４８】

```
BFINS:G:I.W   #src[.BHWL],   offset[.BHWL],   width[.W],   base[.W]
BFINS:G:R.W   Rs[.W],        offset[.BHWL],   width[.W],   base[.W]
BFINS:G:I.L   #src[.BHWL],   offset[.BHWL],   width[.L],   base[.L]
BFINS:G:R.L   Rs[.L],        offset[.BHWL],   width[.L],   base[.L]
BFINS:E:I.W   #src[.BHWL],   #offset,         #width,      base[.W]
BFINS:E:R.W   Rs[.W],        #offset,         #width,      base[.W]
BFINS:E:I.L   #src[.BHWL],   #offset,         #width,      base[.L]
BFINS:E:R.L   Rs[.L],        #offset,         #width,      base[.L]

BFINSU:G:I.W  #src[.BHWL],   offset[.BHWL],   width[.W],   base[.W]
BFINSU:G:R.W  Rs[.W],        offset[.BHWL],   width[.W],   base[.W]
BFINSU:G:I.L  #src[.BHWL],   offset[.BHWL],   width[.L],   base[.L]
BFINSU:G:R.L  Rs[.L],        offset[.BHWL],   width[.L],   base[.L]
BFINSU:E:I.W  #src[.BHWL],   #offset,         #width,      base[.W]
BFINSU:E:R.W  Rs[.W],        #offset,         #width,      base[.W]
BFINSU:E:I.L  #src[.BHWL],   #offset,         #width,      base[.L]
BFINSU:E:R.L  Rs[.L],        #offset,         #width,      base[.L]

BFCMP:G:I.W   #src[.BHWL],   offset[.BHWL],   width[.W],   base[.W]
BFCMP:G:R.W   Rs[.W],        offset[.BHWL],   width[.W],   base[.W]
BFCMP:G:I.L   #src[.BHWL],   offset[.BHWL],   width[.L],   base[.L]
BFCMP:G:R.L   Rs[.L],        offset[.BHWL],   width[.L],   base[.L]
BFCMP:E:I.W   #src[.BHWL],   #offset,         #width,      base[.W]
BFCMP:E:R.W   Rs[.W],        #offset,         #width,      base[.W]
BFCMP:E:I.L   #src[.BHWL],   #offset,         #width,      base[.L]
BFCMP:E:R.L   Rs[.L],        #offset,         #width,      base[.L]
```

【図１４９】

```
BFCMPU:G:I.W  #src[.BHWL],   offset[.BHWL],   width[.W],   base[.W]
BFCMPU:G:R.W  Rs[.W],        offset[.BHWL],   width[.W],   base[.W]
BFCMPU:G:I.L  #src[.BHWL],   offset[.BHWL],   width[.L],   base[.L]
BFCMPU:G:R.L  Rs[.L],        offset[.BHWL],   width[.L],   base[.L]
BFCMPU:E:I.W  #src[.BHWL],   #offset,         #width,      base[.W]
BFCMPU:E:R.W  Rs[.W],        #offset,         #width,      base[.W]
BFCMPU:E:I.L  #src[.BHWL],   #offset,         #width,      base[.L]
BFCMPU:E:R.L  Rs[.L],        #offset,         #width,      base[.L]

BFEXT:G.W     offset[.BHWL], width[.W],       base[.W],    Rd[.W]
BFEXT:G.L     offset[.BHWL], width[.L],       base[.L],    Rd[.L]
BFEXT:E.W     #offset,       #width,          base[.W],    Rd[.W]
BFEXT:E.L     #offset,       #width,          base[.L],    Rd[.L]
BFEXTU:G.W    offset[.BHWL], width[.W],       base[.W],    Rd[.W]
BFEXTU:G.L    offset[.BHWL], width[.L],       base[.L],    Rd[.L]
BFEXTU:E.W    #offset,       #width,          base[.W],    Rd[.W]
BFEXTU:E.L    #offset,       #width,          base[.L],    Rd[.L]
```

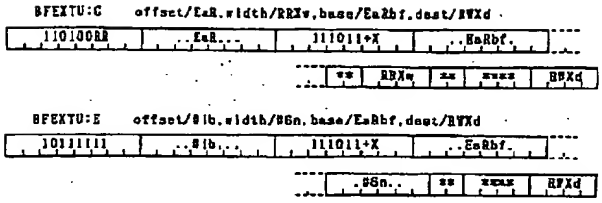
【図１５７】

命令	F	X	V	L	M	Z	コメント
BFINSU	−	−	+	−	+	+	

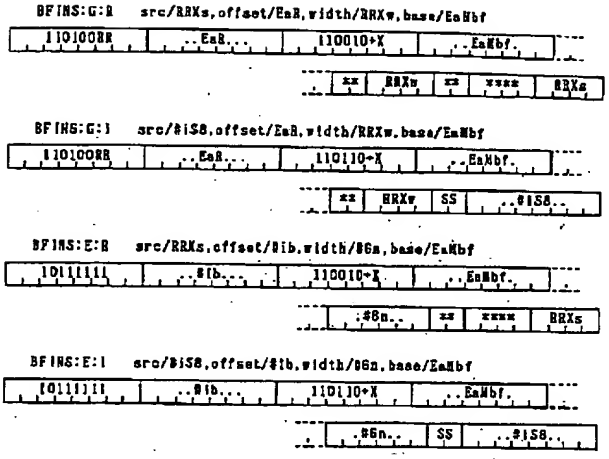
【図１５９】

命令	F	X	V	L	M	Z	コメント
BFCMP	−	−	−	+	−	+	

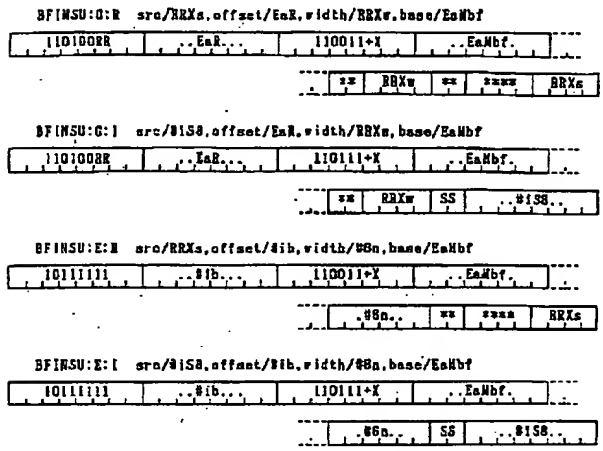
【図１５２】



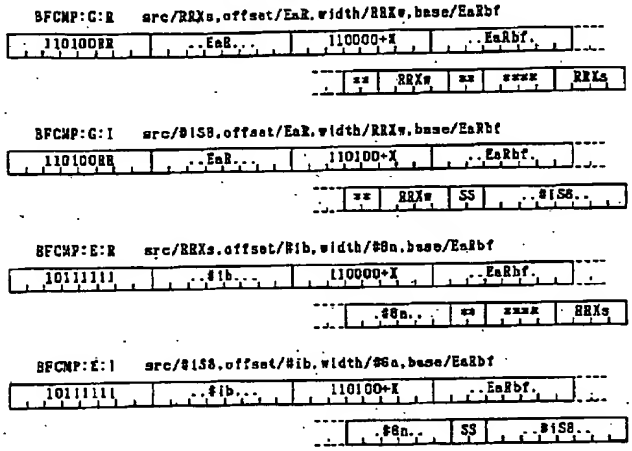
【図１５４】



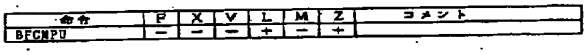
【図１５６】



【図１５８】



【図１６１】

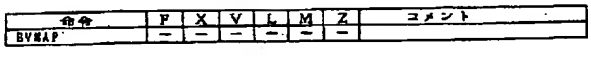


命令	F	X	V	L	M	Z	コメント
BFCMPU	−	−	−	+	−	+	

【図１６３】

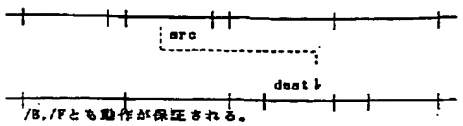
命令	F	X	V	L	M	Z	コメント
BVSCH	−	−	*	−	−	−	Vはサーチ失敗

【図１６５】

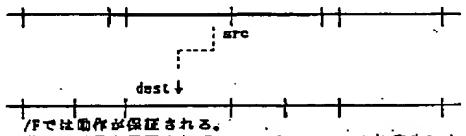


命令	F	X	V	L	M	Z	コメント
BVMAP	−	−	−	−	−	−	

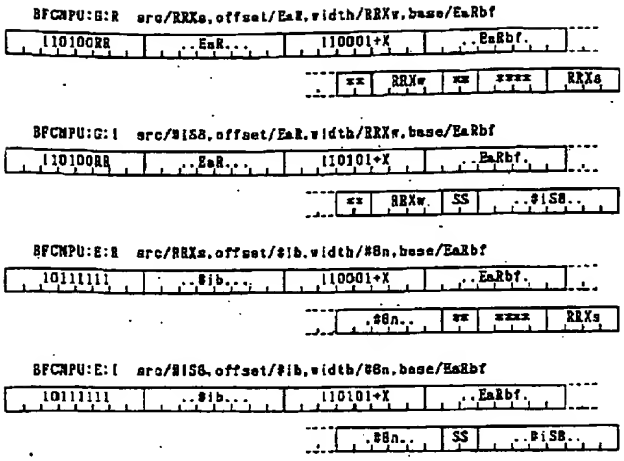
【図１６６】



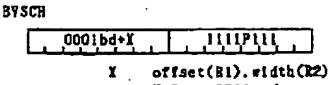
【図１６７】



【図１６０】



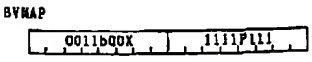
【図１６２】



X　offset(R1),width(R2)のサイズ
X=0　32ビット
X=1　64ビット　<<LX>>
d　サーチするビット値
d=0　'0'をサーチ(/0)
d=1　'1'をサーチ(/1)
b　サーチ方向
b=0　ビット番号増加方向(/F)
b=1　ビット番号減少方向(/B)

レジスタのパラメータ
R0:　base address
R1:　offset
read-modify-writeのオペランドになっており、パラメータである検索開始offsetとリターンパラメータである検索結果のoffsetが入る。dで指定した値のビットが見付かるまで、offsetは何回もワード境界を越える。命令中断時には、その時点で検索中のoffsetが入る。
R2:　width
offsetを検索するビットフィールドの長さ（ビット数）
width(R2)も符号付きとして扱われるが、width≦０の場合は、V_flagのセットだけを行なってそのまま命令を終了する。
EITとはしない。

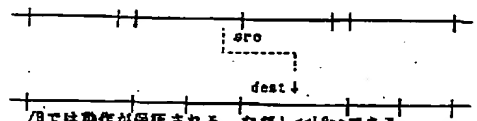
【図１６４】



X　src offset(R1),src width(R2),dest offset(R4)のサイズ
X=0　32ビット
X=1　64ビット　<<LX>>
P　src側のPビットオプション
Q　dest側のPビットオプション
b　処理方向
b=0　ビット番号増加方向(/F)
b=1　ビット番号減少方向(/B)

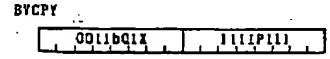
レジスタのパラメータ
R0:　srcのビットフィールドのbase address
R1:　srcのビットフィールドのoffset
符号付きとして扱われ、負の値も許される。
R2:　width
演算するbitfieldの長さ（ビット数）
width(R2)も符号付きとして扱われるが、width≦０の場合は、何もせずに命令を終了する。
EITとはしない。
R3:　destのビットフィールドのbase address
R4:　destのビットフィールドのoffset
符号付きとして扱われ、負の値も許される。
R5:　演算の種類
下位4bitを使用

【図１６８】



/Bでは動作が保証される。ただし<<L2>>である。
/Fでは結果が保証されず、ページフォールトの発生によって結果が変わることもある。

【図１６９】



X　src offset(R1),src width(R2),dest offset(R4)のサイズ
X=0　32ビット
X=1　64ビット　<<LX>>
P　src側のPビットオプション
Q　dest側のPビットオプション
b　処理方向
b=0　ビット番号増加方向(/F)
b=1　ビット番号減少方向(/B)

レジスタのパラメータ
R0:　srcのビットフィールドのbase address
R1:　srcのビットフィールドのoffset
符号付きとして扱われ、負の値も許される。
R2:　width
演算するbitfieldの長さ（ビット数）
width(R2)も符号付きとして扱われるが、width≦０の場合は、何もせずに命令を終了する。
EITとはしない。
R3:　destのビットフィールドのbase address
R4:　destのビットフィールドのoffset
符号付きとして扱われ、負の値も許される。

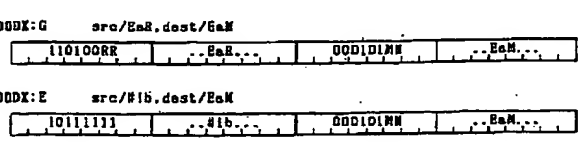
【図１７０】

命令	F	X	V	L	M	Z	コメント
BVCPY	−	−	−	−	−	−	

【図１７２】

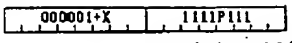
命令	F	X	V	L	M	Z	コメント
BVFAT	−	−	−	−	−	−	

【図１７３】



【図１７１】

BVPAT



X　pattern(R0), width(R2), dest offset(R4)のサイズ
X=0　32ビット
X=1　64ビット　<<LX>>
P　dest側のPビットオプション

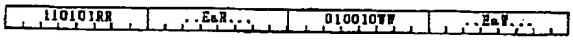
レジスタ上のパラメータ
R0:　pattern
R1:　使用しない
R2:　width
演算するbitfieldの長さ（ビット数）
width(R2)も符号付きとして扱われるが、width≦０の場合は、
何もせずに命令を終了する。
EITとはしない。
R3:　destのビットフィールドのbase address
R4:　destのビットフィールドのoffset
符号付きとして扱われ、負の値も許される。
R5:　演算の種類
下位4bitを使用。BVMAP命令と共通。

【図１７４】

命令	F	X	V	L	M	Z	コメント
ADDDX	−	+	+	0	+	+	

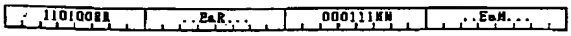
【図１７７】

PACKss　src/EaR,dest/EaW

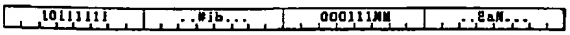


【図１７５】

SUBDX:G　src/EaR,dest/EaM



SUBDX:I　src/#ib,dest/EaM



【図１７６】

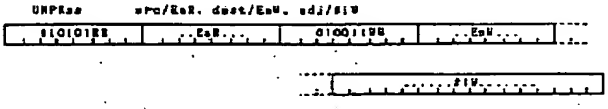
命令	F	X	V	L	M	Z	コメント
SUBDX	−	+	+	+	+	+	

【図１７８】

命令	F	X	V	L	M	Z	コメント
PACKss	−	−	−	−	−	−	

【図１７９】

UNPKss　src/EaR, dest/EaW, adj/#IV

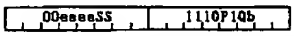


【図１８０】

命令	F	X	V	L	M	Z	コメント
UNPKss	−	−	−	−	−	−	

【図１８２】

SMOV



P　src側のPビットオプション
Q　dest側のPビットオプション
SS　エレメントサイズ、終了条件(R3,R4)のサイズ
b　処理方向
b=0　アドレス増加の方向に処理する(/F)
b=1　アドレス減少の方向に処理する(/B)
eeee　ストリング命令終了条件

レジスタ上のパラメータ
R0:　src側ストリングの先頭アドレス
R1:　dest側ストリングの先頭アドレス
R2:　ストリングの長さ、データ数
R3:　終了条件の比較値(1)
R4:　終了条件の比較値(2)　（ATOMでは使用しない）

【図１８８】

〔フラグ変化〕

命令	F	X	V	L	M	Z	コメント
SSCH	*	−	*	−	*	−	Vはサーチ失敗

【図１８３】

命令	F	X	V	L	M	Z	コメント
SMOV	*	−	*	−	*	−	

【図１８５】

命令	F	X	V	L	M	Z	コメント
SCMP	*	*	*	+	*	+	X,L,Zは最終要素の比較

【図１９２】

命令	F	X	V	L	M	Z	コメント
QINS	−	−	−	−	−	*	

【図１８１】

```
UNPKBH  src(.B),dest(.H),adj(.H)
        RR=00,WW=01     tmpは16ビット
                                0   ==> tmp[00:15],
                          src[00:03] ==> tmp[04:07],
                          src[04:07] ==> tmp[12:15],
                          tmp + adj  ==> dest
UNPKHW  src(.H),dest(.W),adj(.W)        ((1.2))
        RR=01,WW=10     tmpは32ビット
                                0   ==> tmp[00:31],
                          src[00:03] ==> tmp[04:07],
                          src[04:07] ==> tmp[12:15],
                          src[08:11] ==> tmp[20:23],
                          src[12:15] ==> tmp[28:31],

                           tmp + adj  ==> dest
  UNPKBW  src[.B],dest[.W],adj[.W]
          RR=00,WW=10             0   ==> tmp[00:31],
                          src[00:03] ==> tmp[12:15],
                          src[04:07] ==> tmp[28:31],
                          tmp + adj  ==> dest
  UNPKWL  src[.W],dest[.L],adj[.L]      <<LX>>
  UNPKHL  src[.H],dest[.L],adj[.L]      <<LX>>
```

【図１８４】

SCMP

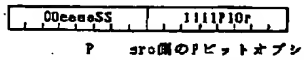
P　src1側のPビットオプション
Q　src2側のPビットオプション
SS　エレメントサイズ、終了条件(R3,R4)のサイズ
b　処理方向
　b=0　アドレス増加の方向に処理する(/F)
　b=1　アドレス減少の方向に処理する(/B)
eeee ストリング命令終了条件

レジスタ上のパラメータ

R0:　src1側ストリングの先頭アドレス
R1:　src2側ストリングの先頭アドレス
R2:　ストリングの長さ、データ数
R3:　終了条件の比較値(1)
R4:　終了条件の比較値(2)　(ATOMでは使用しない)

【図１８７】

SSCH



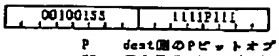
P　src側のPビットオプション
SS　エレメントサイズ、終了条件(R3,R4)のサイズ
eeee ストリング命令終了条件
r　ポインタ更新の方法
　r=0　エレメントサイズだけ増加(/P)
　r=1　増減値をレジスタR5で指定(/B)

レジスタ上のパラメータ

R0:　src側ストリングの先頭アドレス
R1:　使用しない
R2:　ストリングの長さ、データ数
R3:　終了条件の比較値(1)
R4:　終了条件の比較値(2)　(ATOMでは使用しない)
R5:　ポインタ更新値 (/Bオプションの時)

【図１８９】

SSTR



P　dest側のPビットオプション
SS　書き込みデータ(R3)のサイズ

レジスタ上のパラメータ

R0:　使用しない
R1:　dest側ストリングの先頭アドレス
R2:　ストリングの長さ、データ数
R3:　書き込みデータ

【図１８６】

[SCMP終了要因] 長さ	不一致	終了条件	V_flag	Z_flag	L_flag X_flag	F_flag	M_flag
×	×	○(src1=src2)	0	1	0	1	*A
×	○	×(src1)	0	0	*C	0	0+
×	○	○(src1)	0	0	*C	1	*B
○	×	×	1	1	0	0	0+

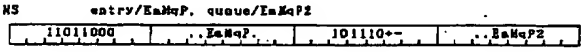
【図１９０】

(フラッグ変化)

命令	F	X	V	L	M	Z	コメント
SSTR	–	–	–	–	–	–	

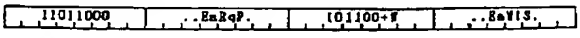
【図１９１】

QINS　　entry/EaMqP, queue/EaMqP2



【図１９６】

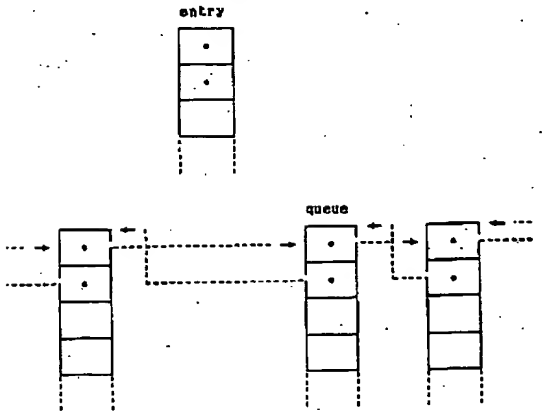
QDEL　　queue/EaRqP, dest/EaW16



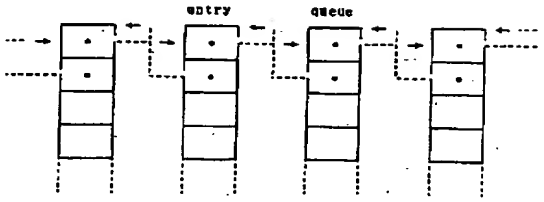
【図１９３】

```
address_of_queue ==> mem[address_of_entry] ==> temp1
mem[address_of_queue + 4] ==> mem[address_of_entry + 4] ==>temp2
address_of_entry ==> mem[mem[address_of_queue + 4]]
address_of_entry ==> mem[address_of_queue + 4]
if (temp1 = temp2) then
      1 ==> Z_flag
else
      0 ==> Z_flag
endif
```

【図１９４】



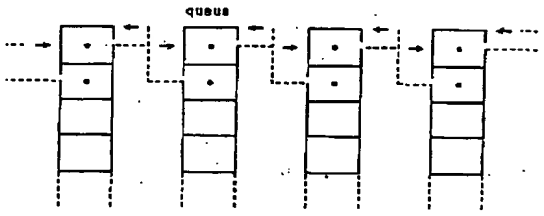
【図１９５】



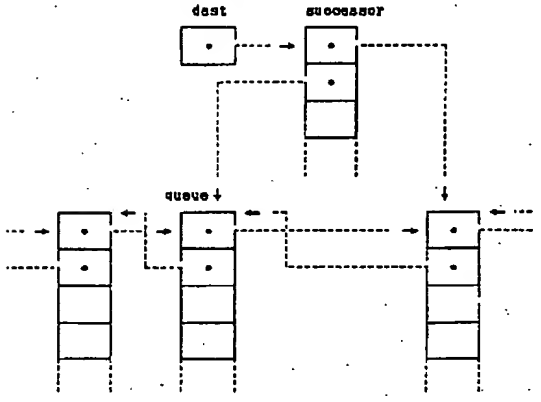
【図１９７】

命令	F	X	V	L	M	Z	コメント
QDEL	−	−	*	−	−	*	

【図１９９】



【図２００】



【図２０１】

QSCH

00eeeeSS	1111P0ab

P　キュー領域のPビットオプション
SS　比較終了条件(R3,R4)とサーチデータのサイズ
eeee 比較終了条件（ストリング命令終了条件と同じ）
a　マスクの有無
　a=0　R6のマスクなし(/NM)
　a=1　R6のマスクあり(/MR)
b　サーチ方向
　b=0　順方向(/F)
　b=1　逆方向(/B)

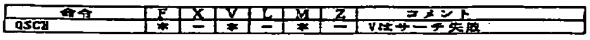
レジスタ上のパラメータ

R0:　サーチを開始するキューエントリのアドレス
　最初はQUEUE HEADの内容 = 最初のエントリのアドレスを入れる。
R1:　リターンパラメータとして使用される。命令終了時に、一つ前のキューエントリのアドレスが入っている。
R2:　キュー終了アドレス
R3:　比較値(1)
R4:　比較値(2)
R5:　エントリ中のサーチデータのオフセット
　サーチするメンバのリンクアドレスからのオフセット
R6:　マスク（M=1の時）

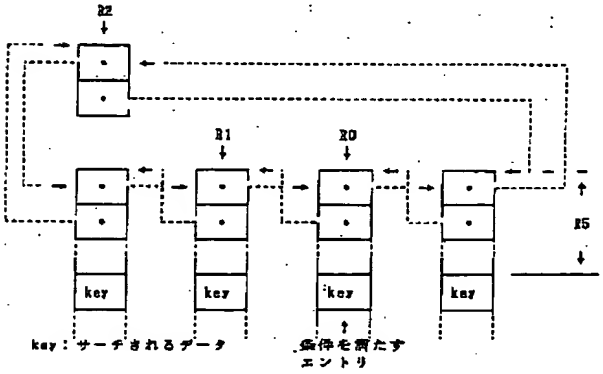
【図１９８】

```
mem[address_of_queue] ==> successor
if(address_of_queue successor) then
        1 ==> V_flag
        1 ==> Z_flag
else
        successor => dest
        mem[successor] ==> mem[address_of_queue] ==> temp1
        address_of_queue ==> mem[mem[successor] + 4] ==> temp2

        if (temp1 = temp2) then
                0 ==> V_flag
                1 ==> Z_flag
        else
                0 ==> V_flag
                0 ==> Z_flag
        endif
endif
```

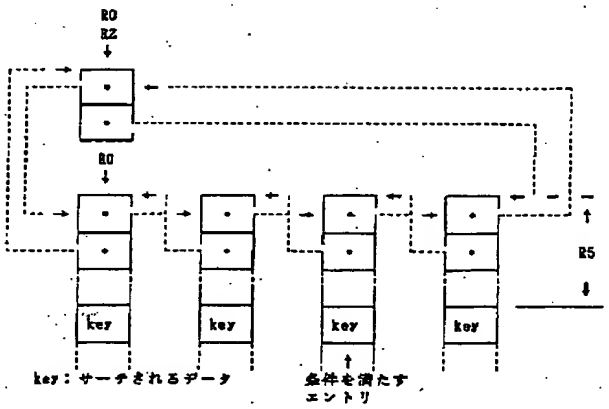
【図２０２】



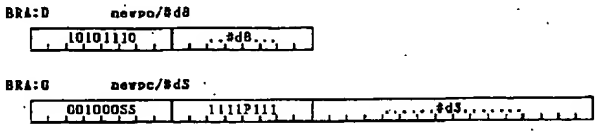
【図２０５】



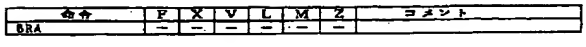
【図２０４】



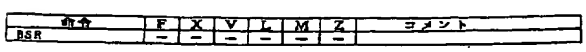
【図２０６】



【図２０７】

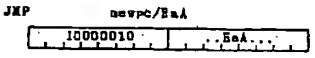


【図２１１】

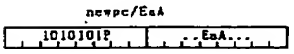


【図２０９】

【図２１２】



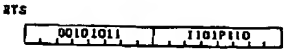
【図２１４】



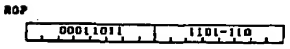
【図２０３】

```
while (1) do
        R0 ==> R1
        if b=0 then
                mem[R0] ==> R0        /*順方向のリンクをたどる*/
        else
                mem[R0+4] ==> R0      /*逆方向のリンクをたどる*/
        if(R0 = R2) then
                1 ==> V_flag
                0 ==> N_flag
                0 ==> F_flag
                exit                  /*サーチ失敗*/
        endif
        if m=0 then
                compare mem[R0+R5] with R3, R4
                and set F_flag, N_flag according to eeee
                        /*終了条件成立の場合F_flag=1となる*/
        else
                compare (mem[R0+R5] & R6) with R3, R4
                and set F_flag, N_flag according to eeee
                        /*終了条件成立の場合F_flag=1となる*/
        endif
        if (F_flag = 1) then
                0 ==> V_flag
                exit                  /*サーチ成功*/
        endif
        check_interrupt
end_while
```

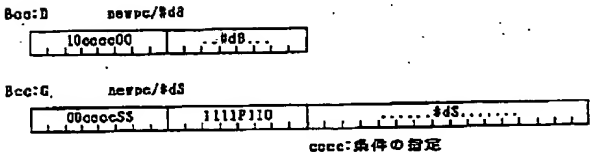
【図２２７】



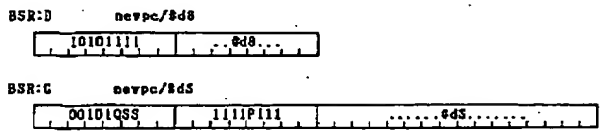
【図２２９】



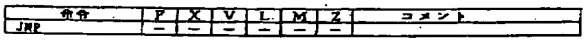
【図２０８】



【図２１０】



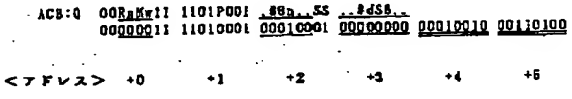
【図２１３】



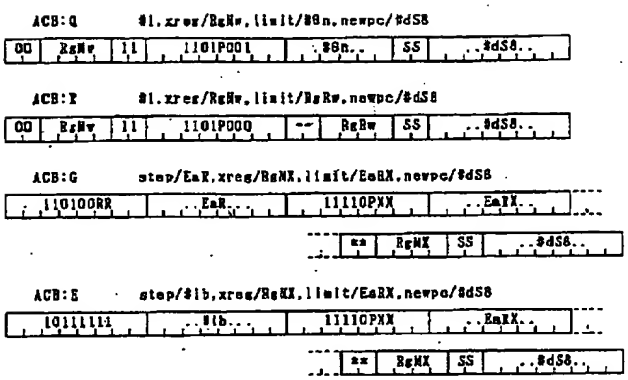
【図２１５】

【図２１７】

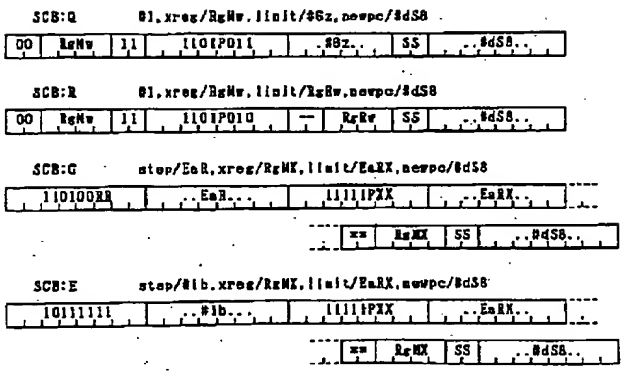
【図２１８】



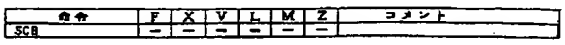
【図２１６】



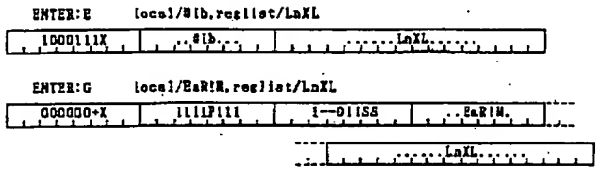
【図２１９】



【図２２０】

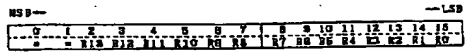


【図２２１】

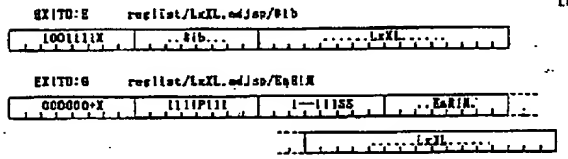


【図２２２】

【図２２３】

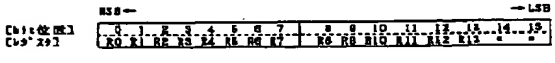


【図２２４】



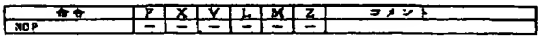
【図２２５】

【図２２６】

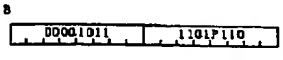


【図２２８】

【図２３０】

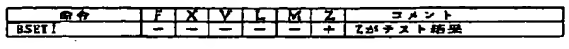


【図２３１】

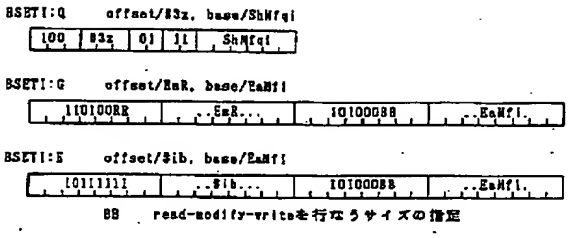


【図２３２】

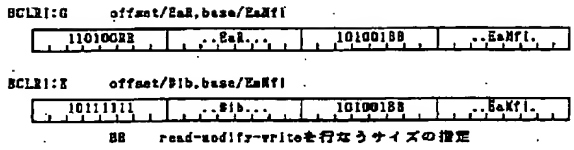
【図２３４】



【図２３３】

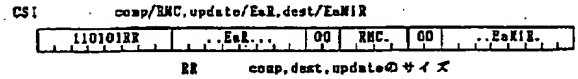


【図２３５】

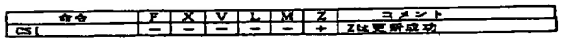


【図２３６】

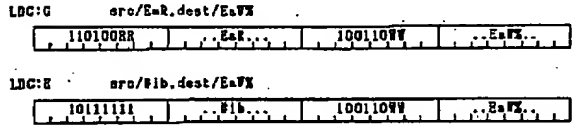
【図２３７】



【図２３８】

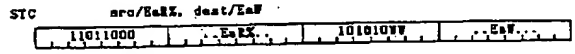


【図２３９】



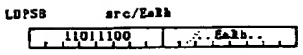
【図２４０】

【図２４１】

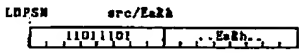


【図２４２】

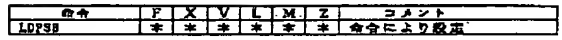
【図２４３】



【図２４５】

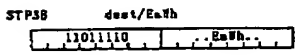


【図２４４】



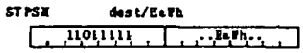
【図２４６】

【図２４７】

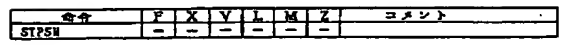


【図２４８】

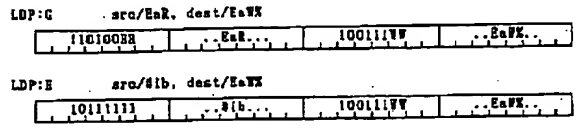
【図２４９】



【図２５０】

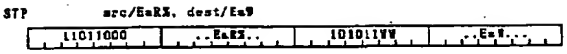


【図２５１】

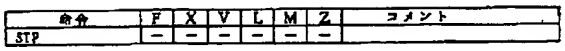


【図２５２】

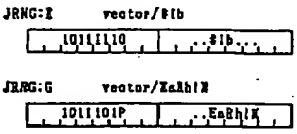
【図２５３】



【図２５４】

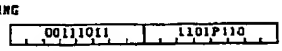


【図２５５】

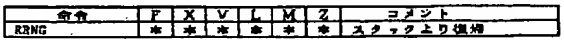


【図２５６】

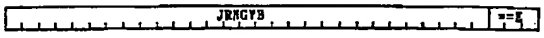
【図２６３】



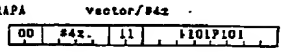
【図２６４】



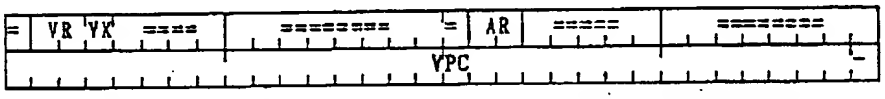
【図２５７】



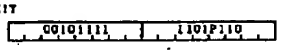
【図２６８】



【図２５８】

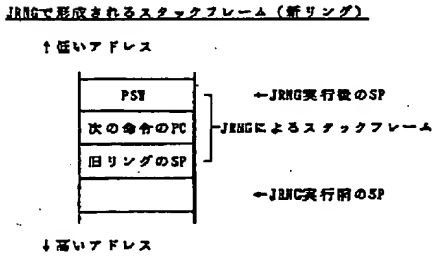


【図２７２】

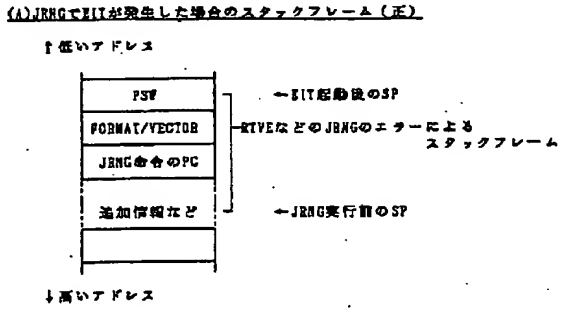


VR(Vector RNG):JRNG命令実行によって新しく移行する先のリング番号
AR(Access RNG):JRNG命令の実行が許可される最も外側のリング番号
VX:(Vector XA):JRNG命令実行後の新XA
現在は０に固定
VPC(Vector PC):JRNG命令実行後の新PC

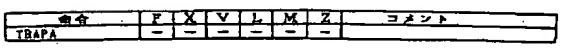
【図２５９】



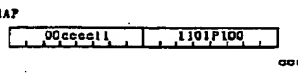
【図２６０】



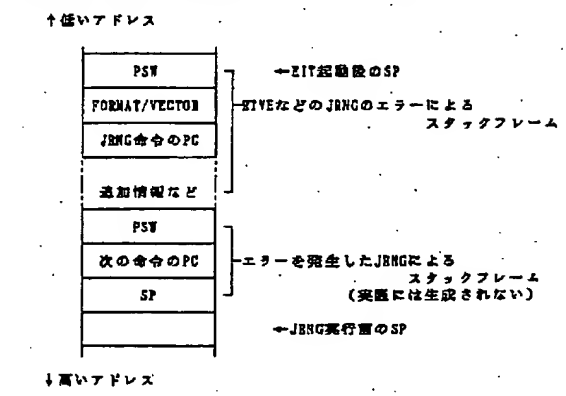
【図２６９】



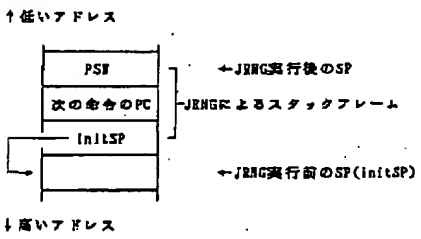
【図２７０】



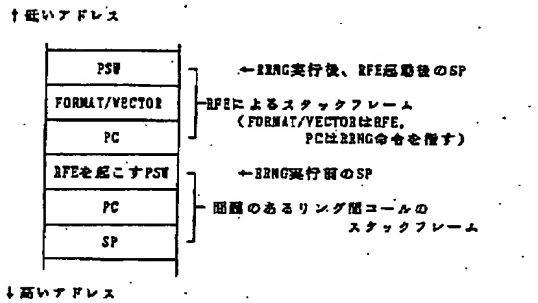
【図２６１】



【図２６２】

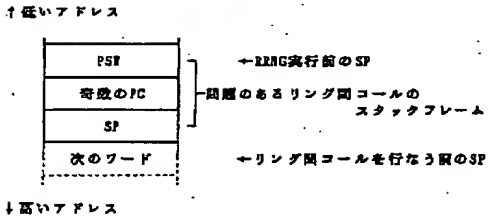


【図２６５】

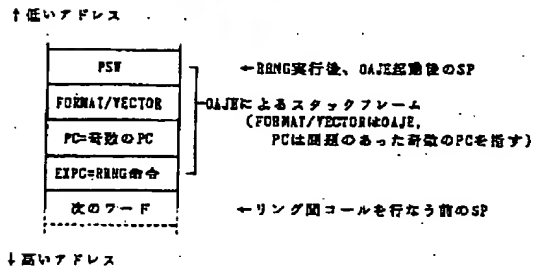


※リング間コールを行なった側のSPは元のままである。
※PSWは、RRNG命令実行前のPSWの値を、RFE,RTVEのEITVTEによって書き換えたものになる。PRNGは、RRNG命令を実行したリングを指す。スタック中に退避されていたPSW（RFEを起こすPSW）は、EIT起動後のPSWには影響しない。

【図２６６】



【図２６７】



※リング間コールを行なった側のSPは、スタックから復帰されたものになる。
※PSWは、一旦スタックから復帰した値を、OAJEのEITVTEによって書き換えたものになる。PRNGは、リング間コールを行なったリングを指す。つまり、ソフトウェアによってPRNGを書き換えていない限り、RRNG命令実行前のPRNGの値と同じである。

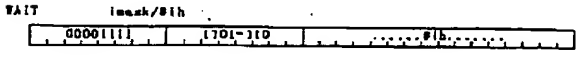
【図２７３】

命令	F	X	V	L	M	Z	コメント
REIT	*	*	*	*	*	*	スタックより復帰

【図２７１】

命令	F	X	V	L	M	Z	コメント
TRAP	-	-	-	-	-	-	

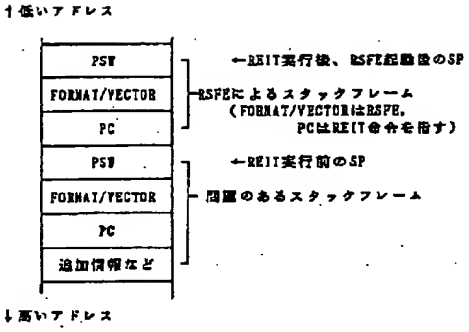
【図２７５】



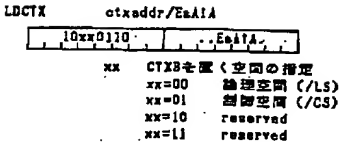
【図２７６】

命令	F	X	V	L	M	Z	コメント
WAIT	-	-	-	-	-	-	

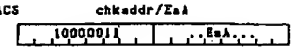
【図２７４】



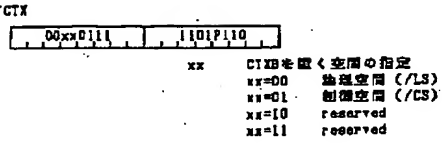
【図２７７】



【図２８１】



【図２７９】



【図２７８】

命令	F	X	V	L	M	Z	コメント
LDCTX	—	—	—	—	—	—	

【図２８２】

命令	F	X	V	L	M	Z	コメント
ACS	—	—	—	*	*	*	

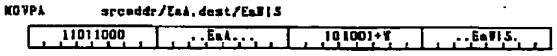
【図２８０】

命令	F	X	V	L	M	Z	コメント
STCTX	—	—	—	—	—	—	

【図２８４】

命令	F	X	V	L	M	Z	コメント
MOVPA	*	—	*	—	—	—	

【図２８３】



レジスタ上のパラメータ
R1　アドレス変換テーブルのベースアドレス

【図２８６】

アドレス	一般のメモリアクセス	MOVPA,LDATE,STATE命令
H'00000000—7fffffff	UATBによりアドレス変換	R1によりアドレス変換
H'80000000—ffffffff	SATBによりアドレス変換	SATBによりアドレス変換

【図２８５】

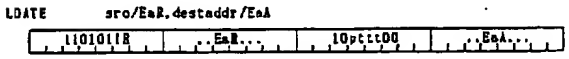
	V_flag	P_flag	結果
正常終了	0	0	物理アドレス ==> dest
エラー	1	0	destは変化しない
ページアクト(ST,PT,PAGE)	1	1	destは変化しない

エラーには、ATEのフォーマットエラー（予約ATEエラー）や、ATEによって未使用領域の指定があった場合が含まれる。

【図２８８】

命令	F	X	V	L	M	Z	コメント
LDATE	*	—	*	—	—	—	

【図２８７】



p　ページを行なう論理空間の指定
p=0　すべての空間(/AS)
p=1　R0で指定したLSIDを持つ空間(/SS)
ttt　ロードを行なうATEの指定
ttt=000 PTEへのロード
ttt=001 STEへのロード

レジスタ上のパラメータ
R0　ページを行なうTLBの論理空間のLSID（/SSの時のみ）
R1　アドレス変換テーブルのベースアドレス

【図２８９】

	V_flag	F_flag	結果
正常終了	0	0	ATEはセットされる
ロードしたATEのP1=0	0	1	ATEはセットされる
ロードしたATEの予約ATEエラー	0	1	ATEはセットされる
途中段ATEの予約ATEエラー	1	0	ATEはセットされない
途中段ATEのP1=0（ページ不在）	1	1	ATEはセットされない

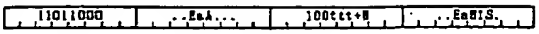
V_flagは予約ATEエラーやページ不在によって、ATEへの設定ができなかったことを示す

【図２９０】

命令	F	X	V	L	M	Z	コメント
STATE	*	-	*	-	-	-	

【図２９１】

STATE　　srcaddr/EaA,dest/EaVIS



ttt　ストアを行なうATEの指定
ttt=000 PTEからのストア
ttt=001 STEからのストア

レジスタ上のパラメータ
R1　アドレス変換テーブルのベースアドレス

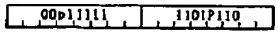
【図２９２】

	V_flag	F_flag	結果
正常終了	0	0	ATE ==> dest
読み出されたATEのPI=0	0	1	ATE ==> dest
読み出されたATEの予約ATEエラー	0	1	ATE ==> dest
途中段のATEの予約ATEエラー	1	0	destは変化しない
途中段のページアウト	1	1	destは変化しない

V_flagは、予約ATEエラーやページアウトによって、ATEの読みだしができなかったことを示す。
なお、予約ATEエラーは、ATEのreservedのビットを使用していた場合などに発生するものである。

【図２９３】

PTLB



p　ページを行なう論理空間の指定
p=0　すべての空間(/AS)
p=1　RDで指定したLSIDを持つ空間(/SS)

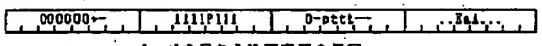
レジスタ上のパラメータ
RD　ページを行なうTLBの論理空間のLSID(/SSの時のみ)

【図２９４】

命令	F	X	V	L	M	Z	コメント
PTLB	-	-	-	-	-	-	

【図２９５】

PSTLB　　prgaddr/EaA



p　ページを行なう論理空間の指定
p=0　すべての空間(/AS)
p=1　RDで指定したLSIDを持つ空間(/SS)
ttt　ページを行なう論理アドレス範囲の指定
ttt=000 論理アドレス全体(2^31～2^12ビット)が一致するエントリをページ(/PT)
ttt=001 論理アドレスの2^31～2^22ビットが一致するエントリをページ(/ST)
ttt=110 論理アドレスの2^31ビットが一致するエントリをページ(/AT)

レジスタ上のパラメータ
RD　ページを行なうTLBの論理空間のLSID(/SSの時のみ)

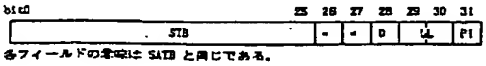
【図２９６】

命令	F	X	V	L	M	Z	コメント
PSTLB	-	-	-	-	-	-	

【図２９７】

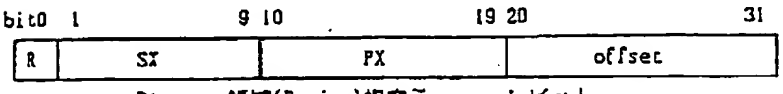
AT	意味
00	アドレス変換、メモリ保護なし
01	reserved
10	アドレス変換なし、アドレスのみを使った簡単なメモリ保護 <<LIB>>（アドレスのMSBによるメモリ領域の区別、2リング）
11	reserved

【図３０１】



各フィールドの意味は SATB と同じである。

【図２９８】



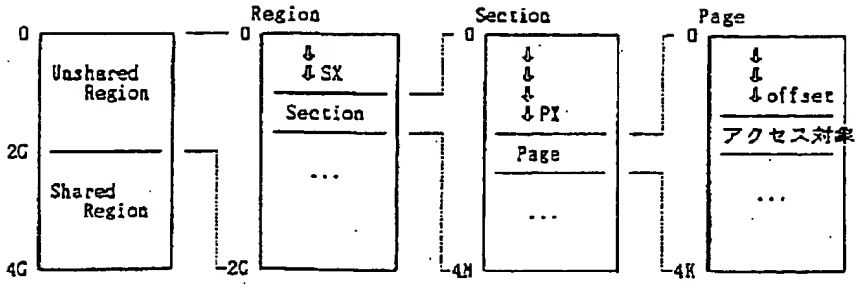
R:　領域(Region)指定子　1 ビット
SX:　Section index　9 ビット
PX:　Page index　10 ビット
offset:　Page offset　12 ビット

【図３０８】

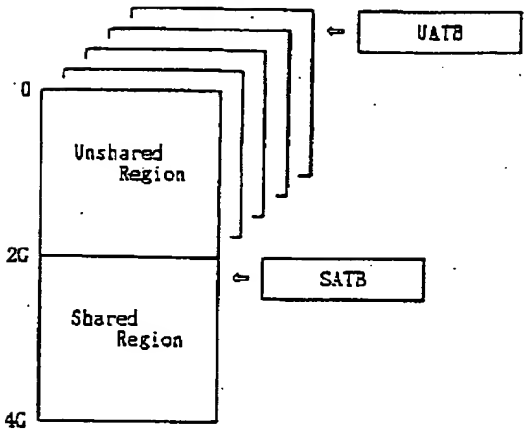
D	LL	使用できるSXの値の範囲(2進ビット対応)	
0	00	** **** ****	512 エントリ
0	01	0* **** ****	256 エントリ
0	10	00 0000 ****	64 エントリ
0	11	00 0000 ****	16 エントリ
1	00	(別途参照)	
1	01	11 **** ****	256 エントリ
1	10	11 11** ****	64 エントリ
1	11	11 1111 ****	16 エントリ

'*' は 0/1 の両方の値をとれることを示す

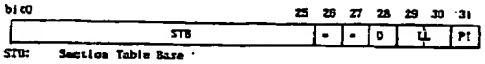
【図２９９】



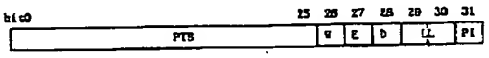
【図３００】



【図３０２】



【図３０７】

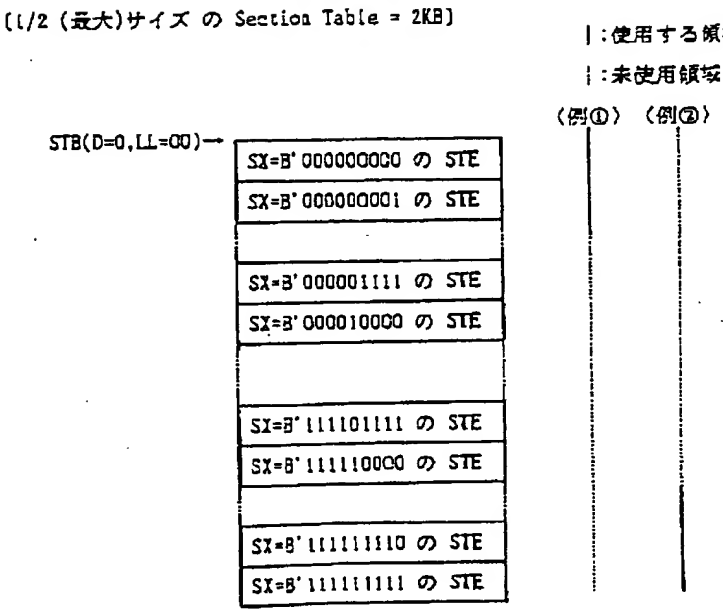


【図３０３】

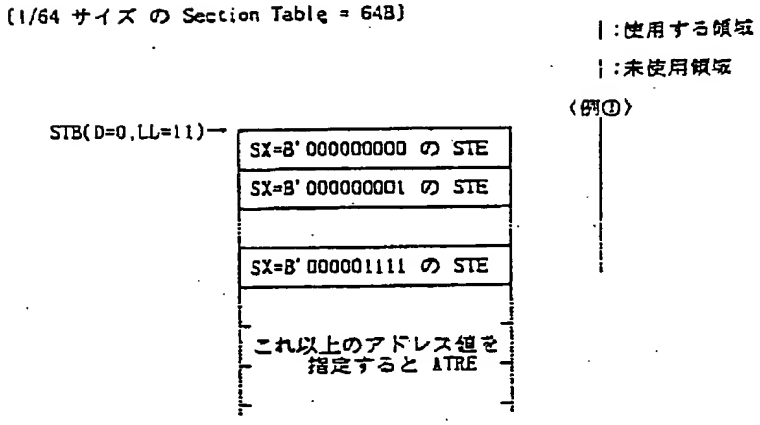
D	LL	使用できるSXの値の範囲(2進ビット対応)	
=	00	* * * * * * * * * *	512 エントリ
0	01	0 * * * * * * * * *	256 エントリ
0	10	0 0 0 * * * * * * *	64 エントリ
0	11	0 0 0 0 0 * * * * *	16 エントリ
1	01	1 * * * * * * * * *	256 エントリ
1	10	1 1 1 * * * * * * *	64 エントリ
1	11	1 1 1 1 1 * * * * *	16 エントリ

'=' は 0/1 の両方の値をとれることを示す
'=' は '0' に reserved であるが、インプリメント上は正反時も無視する
ことを示す.

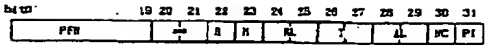
【図３０４】



【図３０５】



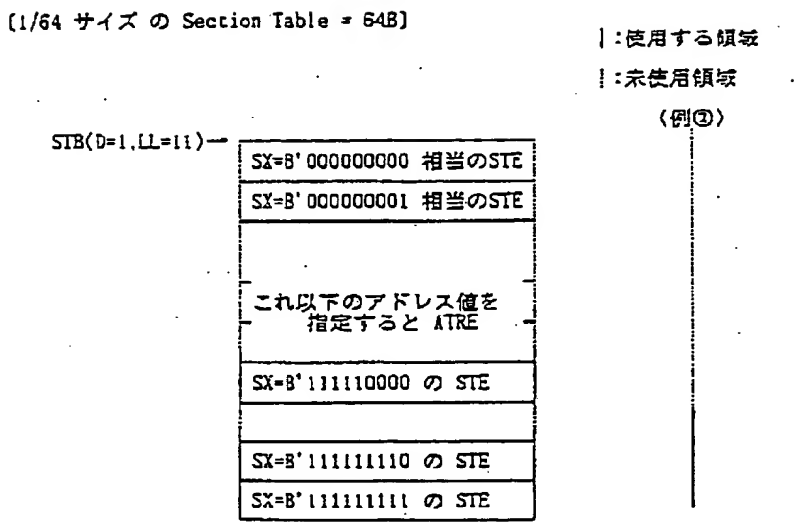
【図３１１】



【図３０９】

D	LL	W E	動作
1	00	0 *	アドレス変換例外(ATRE) の 未使用領域参照エラーを発生
1	00	1 *	(reserved) アドレス変換例外(ATRE) の 予約ATEエラーを発生

【図３０６】

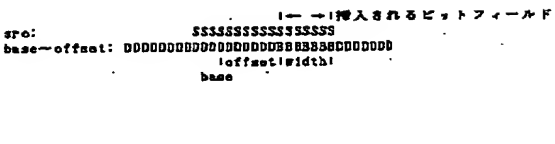


【図３１０】

bit0	10	20	32	
R(==SX===)(===PX===)(==offset==)				ATB, STE の LL=00
R-(==SX==)—(==PX==)(==offset==)				ATB, STE の LL=01
R——(=SX=)——(=PX=)(==offset==)				ATB, STE の LL=10
R———(SX)———(PX)(==offset==)				ATB, STE の LL=11

-: アドレスが 0 または 1 に固定される範囲

【図３２３】



【図３１３】

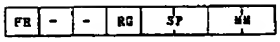
T	AL	RL	動作
00	00	??	RL は本来の意味で使用
00	01	**	アドレス変換例外(ATRE) の 未使用領域参照エラーを発生
00	10	??	(reserved) アドレス変換例外(ATRE) の 予約ATEエラーを発生
00	11	??	(reserved) アドレス変換例外(ATRE) の 予約ATEエラーを発生

'*' はソフトウエアで自由に使用してよいビットである。ハードウエア的にはこのビットが無視される。

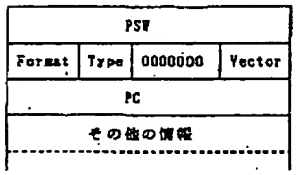
【図３１２】

		T=00				T=01				T=10				T=11			
AL	RL	R0	R1	R2	R3	R0	R1	R2	R3	R0	R1	R2	R3	R0	R1	R2	R3
00	00	R—	—	—	—	R-E	—	—	—	RW-	—	—	—	RWE	—	—	—
00	01	R—	R—	—	—	R-E	R-E	—	—	RW-	R—	—	—	RWE	R-E	—	—
00	10	R—	R—	R—	—	R-E	R-E	R-E	—	RW-	R—	R—	—	RWE	R-E	R-E	—
00	11	R—	R—	R—	R—	R-E	R-E	R-E	R-E	RW-	R—	R—	R—	RWE	R-E	R-E	R-E
01	00	別表参照				R-E	—E	—	—	RW-	—	—	—	RWE	—E	—	—
01	01					R-E	R-E	—	—	RW-	RW-	—	—	RWE	RWE	—	—
01	10					R-E	R-E	R-E	—	RW-	RW-	R—	—	RWE	RWE	R-E	—
01	11					R-E	R-E	R-E	R-E	RW-	RW-	R—	R—	RWE	RWE	R-E	R-E
10	00	別表参照				R-E	—E	—E	—	RW-	—	—	—	RWE	—E	—E	—
10	01					R-E	R-E	—E	—	RW-	RW-	—	—	RWE	RWE	—E	—
10	10					R-E	R-E	R-E	—	RW-	RW-	RW-	—	RWE	RWE	RWE	—
10	11					R-E	R-E	R-E	R-E	RW-	RW-	RW-	R—	RWE	RWE	RWE	R-E
11	00	別表参照				R-E	—E	—E	—E	RW-	—	—	—	RWE	—E	—E	—E
11	01					R-E	R-E	—E	—E	RW-	RW-	—	—	RWE	RWE	—E	—E
11	10					R-E	R-E	R-E	—E	RW-	RW-	RW-	—	RWE	RWE	RWE	—E
11	11					R-E	R-E	R-E	R-E	RW-	RW-	RW-	RW-	RWE	RWE	RWE	RWE

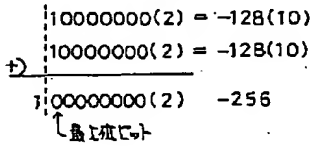
【図３５７】



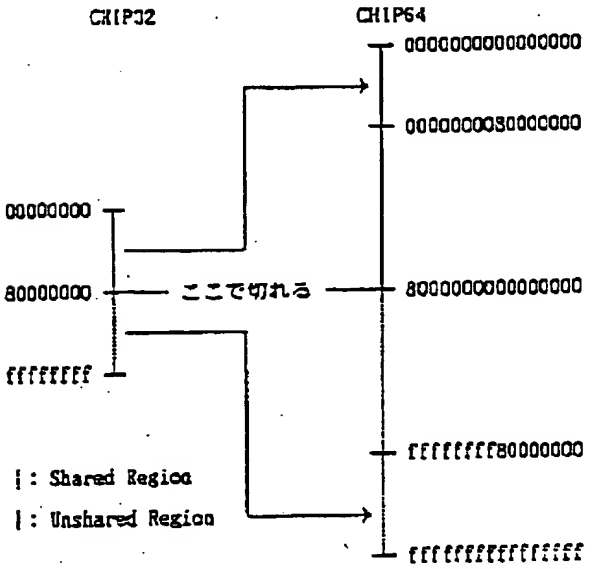
【図３６０】



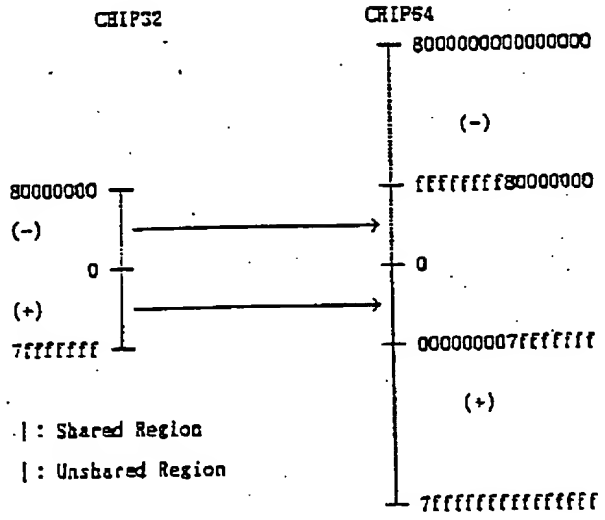
【図４１６】



【図３１４】



【図３１５】



【図３１６】

命令	F	X	V	L	M	Z	コメント
CMP	—	—	—	+	—	+	
CMPU	—	—	—	+	—	+	
CHK	—	—	*	+	—	+	L,Zは下限値との比較

【図３１７】

命令	F	X	V	L	M	Z	コメント
MOV	−	−	+	−	+	+	
MOVU	−	−	+	−	+	+	
PUSH	−	−	−	−	−	−	
POP	−	−	−	−	−	−	
STM	−	−	−	−	−	−	
LDM	−	−	−	−	−	−	
MOVA	−	−	−	−	−	−	
PUSHA	−	−	−	−	−	−	

【図３１８】

命令	F	X	V	L	M	Z	コメント
ADD	−	+	+	+	+	+	
ADDU	−	+	+	0	+	+	
ADDX	−	+	+	+	+	+	
SUB	−	+	+	+	+	+	
SUBU	−	+	+	+	+	+	
SUBX	−	+	+	+	+	+	
MUL	−	−	+	+	+	+	
MULU	−	−	+	0	+	+	
MULX	*	−	0	0	+	+	N,Zはdest基準、Fはtmp=0
DIV	−	−	0	+	+	+	
	→	−	1	0	1	0	-(最小負数)÷(-1)の場合
	−	−	1	−	−	−	-0除算，EITの場合
DIVU	−	−	0	0	+	+	
	−	−	1	−	−	−	-0除算，EITの場合
DIVX	*	−	0	0	+	+	N,Zはdest基準、Fはtmp=0
	−	−	1	−	−	−	-dest=A'-フローの場合
	−	−	1	−	−	−	-0除算，EITの場合
REM	−	−	0	+	+	+	
	−	−	0	−	−	−	-0除算，EITの場合
REMU	−	−	0	0	+	+	
	−	−	0	−	−	−	-0除算，EITの場合
NEG	−	−	+	+	+	+	
INDEX	−	−	+	+	+	+	N,Zはxreg基準

【図３１９】

命令	F	X	V	L	M	Z	コメント
AND	−	−	−	−	+	+	
OR	−	−	−	−	+	+	
XOR	−	−	−	−	+	+	
NOT	−	−	−	−	+	+	

【図３２０】

命令	F	X	V	L	M	Z	コメント
SHL	−	+	−	−	+	+	
SHA	−	+	+	+	+	+	
ROT	−	+	−	−	+	+	
SHXL	−	+	−	−	+	+	
SHXR	−	+	−	−	+	+	
RVBY	−	−	−	−	−	−	
RVBI	−	−	−	−	−	−	

【図３２１】

命令	F	X	V	L	M	Z	コメント
BTST	−	−	−	−	−	+	Zがテスト結果
BSET	−	−	−	−	−	+	Zがテスト結果
BCLR	−	−	−	−	−	+	Zがテスト結果
BNOT	−	−	−	−	−	+	Zがテスト結果
BSCH	−	−	*	−	−	−	Vはサーチ失敗

【図３２２】

命令	F	X	V	L	M	Z	コメント
BFEXT	−	−	+	−	+	+	
BFEXTU	−	−	+	−	+	+	
BFINS	−	−	+	−	+	+	
BFINSU	−	−	+	−	+	+	
BFCMP	−	−	−	+	−	+	
BFCMPU	−	−	−	+	−	+	

【図３２４】

命令	F	X	V	L	M	Z	コメント
BVSCB	−	−	*	−	−	−	Vはサーチ失敗
BVMAP	−	−	−	−	−	−	
BVCPY	−	−	−	−	−	−	
BVPAT	−	−	−	−	−	−	

【図３２５】

命令	F	X	V	L	M	Z	コメント
ADDDX	−	+	+	0	+	+	
SUBDX	−	+	+	+	+	+	
PACKss	−	−	−	−	−	−	
UNPKss	−	−	−	−	−	−	

【図３２６】

命令	F	X	V	L	M	Z	コメント
SMOV	*	−	*	−	*	−	
SCMP	*	*	*	+	*	+	X,L,Zは最終要素の比較
SSCH	*	−	*	−	*	−	Vはサーチ失敗
SSTB	−	−	−	−	−	−	

【図３２７】

命令	F	X	V	L	M	Z	コメント
QINS	−	−	−	−	−	*	
QDEL	−	−	*	−	−	*	
QSCH	*	−	*	−	*	−	Vはサーチ失敗

【図３２９】

命令	F	X	V	L	M	Z	コメント
BSETI	−	−	−	−	−	+	Zがテスト結果
BCLRI	−	−	−	−	−	+	Zがテスト結果
CSI	−	−	−	−	−	+	Zは更新成功

【図３２８】

命令	F	X	V	L	M	Z	コメント
BRA	－	－	－	－	－	－	
Bcc	－	－	－	－	－	－	
BSR	－	－	－	－	－	－	
JMP	－	－	－	－	－	－	
JSR	－	－	－	－	－	－	
ACB	－	－	－	－	－	－	
SCB	－	－	－	－	－	－	
ENTER	－	－	－	－	－	－	
EXITD	－	－	－	－	－	－	
RTS	－	－	－	－	－	－	
NOP	－	－	－	－	－	－	
PIB	－	－	－	－	－	－	

【図３３０】

命令	F	X	V	L	M	Z	コメント
LDC	－	－	－	－	－	－	
	＊	＊	＊	＊	＊	＊	←dst=PSRの場合
STC	－	－	－	－	－	－	
LDPSB	＊	＊	＊	＊	＊	＊	命令により設定
LDPSW	－	－	－	－	－	－	
STPSB	－	－	－	－	－	－	
STPSW	－	－	－	－	－	－	
LDP	－	－	－	－	－	－	
STP	－	－	－	－	－	－	

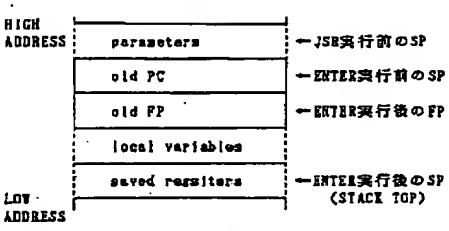
【図３３２】

命令	F	X	V	L	M	Z	コメント
ACS	－	－	－	＊	＊	＊	
MOVPA	＊	－	＊	－	－	－	
LDATE	＊	－	＊	－	－	－	
STATE	＊	－	＊	－	－	－	
PTLB	－	－	－	－	－	－	
PSTLB	－	－	－	－	－	－	
PLCH	－	－	－	－	－	－	
PSLCH	－	－	－	－	－	－	

【図３３１】

命令	F	X	V	L	M	Z	コメント
JRNG	－	－	－	－	－	－	
RRNG	＊	＊	＊	＊	＊	＊	スタックより復帰
TRAPA	－	－	－	－	－	－	
TRAP	－	－	－	－	－	－	
REIT	＊	＊	＊	＊	＊	＊	スタックより復帰
WAIT	－	－	－	－	－	－	
LDCTX	－	－	－	－	－	－	
STCTX	－	－	－	－	－	－	

【図３３４】



【図３３３】

＜サブルーチンの入口＞

1.PCの退避と
新しいPCの設定

2.FPの退避と
新しいFPの設定

3.ローカル変数の
領域の確保

4.レジスタの退避

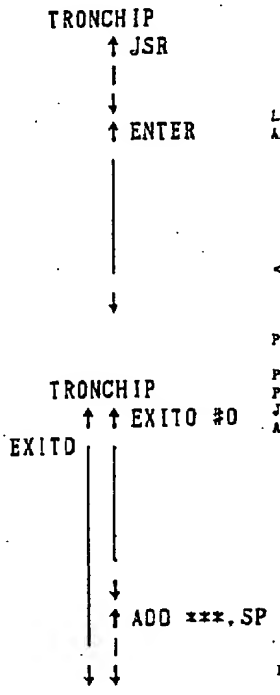
＜サブルーチンの出口＞

5.レジスタの復帰

6.ローカル変数解放
とFPの復帰

7.PCの復帰と
リターン

8.スタック上の
パラメータ解放



【図３３５】

＜呼出側＞

```
PUSH  paraN
PUSH  para2
PUSH  para1
JSR   sub
ADD   #N*4,SP
```

＜呼ばれる側＞

```
sub:  ENTER  #area,reglist
      |パラメータと局所変数は
      |d(disp,FP)により アクセス
      EXITD reglist,#0
```

【図３３６】

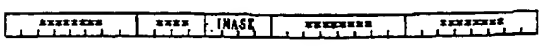
＜呼出側＞

```
PUSH  paraN
PUSH  para2
PUSH  para1
JSR   sub
```

＜呼ばれる側＞

```
sub:  ENTER  #area,reglist
      |パラメータと局所変数は
      |d(disp,FP)によりアクセス
      EXITD reglist,#N*4
```

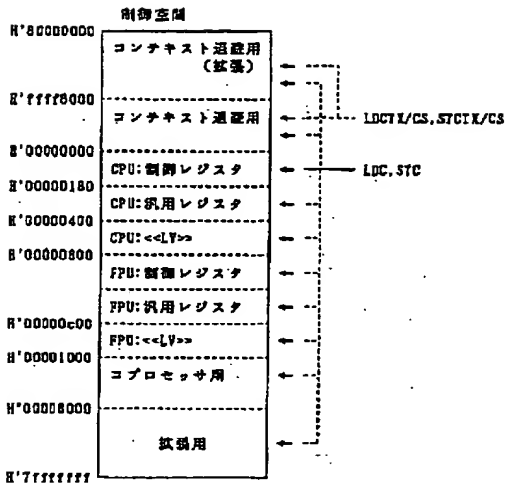
【図３４１】



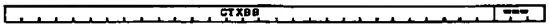
【図３３７】

```
procedure proc0 {レキシカルレベル０} ;
  var var0;
  procedure proc1A {レキシカルレベル１} ;
    var var1A;
    procedure proc2A {レキシカルレベル２} ;
      var var2A;
      begin
        ...
      end
    procedure proc2B {レキシカルレベル２} ;
      var var2B;
      begin
        ...
      end
    begin {procedure proc1A}
      ...
    end
  procedure proc1B {レキシカルレベル１} ;
    var var1B;
    procedure proc2C {レキシカルレベル２} ;
      var var2C;
      begin
        ...
      end
    procedure proc2D {レキシカルレベル２} ;
      var var2D;
      begin
        ...
      end
    begin {procedure proc1B}
      ...
    end
  begin {procedure proc0}
    ...
  end
```

【図３３９】

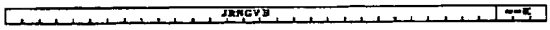


【図３４３】



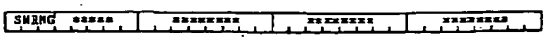
【図３４６】

【図３４９】

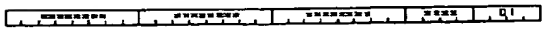


【図３４０】

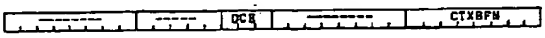
【図３４２】



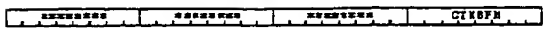
【図３４４】



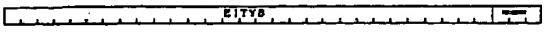
【図３４５】



【図３４７】



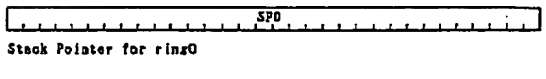
【図３４８】



【図３３８】

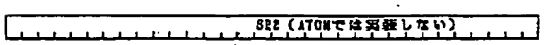
<<サブルーチンコール状態>>	<<ディスプレイ>>				
	FP lev=X	R13 lev=0	R12 lev=1	R11 lev=2	R10 lev=3
proc0	proc0 (var0)	proc0	-	-	-
↓					
[lev. up, FP -> R12]					
proc1A	proc1A (var1A)	proc0 (var0)	proc1A		
↓					
[lev. up, FP -> R11]					
proc2B	proc2B (var2B)	proc0 (var0)	proc1A (var1A)	proc2B	
↓					
[lev. same, FP -> R11]					
proc2A	proc1A (var2A)	proc0 (var0)	proc1A (var1A)	proc2A	
↓					
[lev. down, FP -> R13]					
proc0(recursion)	proc0* (var0*)	proc0*	-	-	
↓					
[lev. up, FP -> R12]					
proc1B	proc1B (var1B)	proc0* (var0*)	proc1B	-	
↓					
[lev. up, FP -> R11]					
proc2D	proc2D (var2D)	proc0* (var0*)	proc1B (var1B)	proc2D	
↓					
...					

【図３５０】

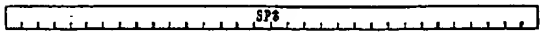


Stack Pointer for ring0

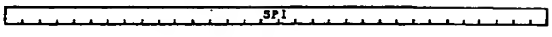
Stack Pointer for ring1



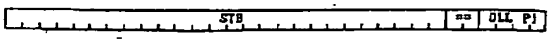
Stack Pointer for ring2



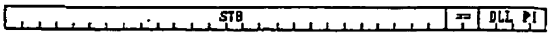
【図３５１】



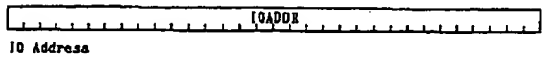
【図３５３】



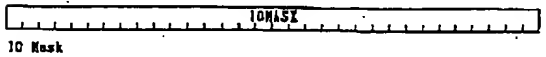
【図３５４】



【図３５２】

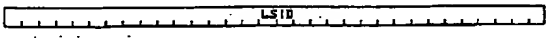


IO Address



IO Mask

【図３５５】



【図３５６】

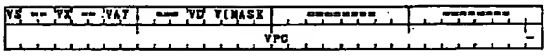
低アドレス↑

<<<スタックをそのまま使う>>>

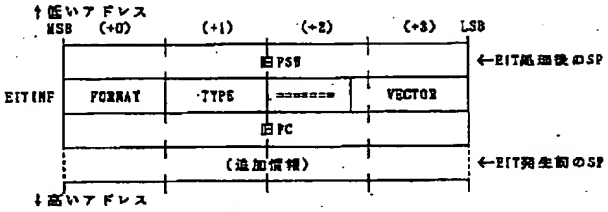
CTXB ⟶ CSW [32bit]
SP0 [32bit] ⟶ PSW[32bit]
SP1 [32bit] *1,*2　FORMAT/VECTOR[32bit]
SP2 [32bit] *1,*2　PC[32bit]
SP3 [32bit] *1　...
UATB[32bit] *2　(EIT追加情報) [32×n bit]
LSID[32bit] *3　...
R0 [32bit] *4
...
R14 [32bit] *4
...
(コプロセッサレジスタ退避用) *5
...
(OS用 - Process ID,Task IDなど)
...

高アドレス↓

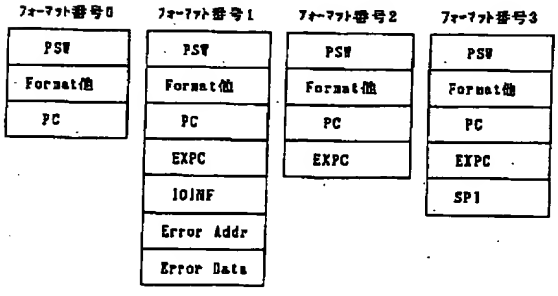
【図３５８】



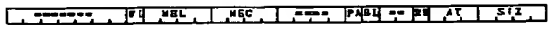
【図３５９】



【図３６１】



【図３６２】



【図３６８】

IMASK	禁止される DI	許可される外部割り込み
0	—	INT 0 (NMI)
1	DI 0	INT 0 (NMI)
2	DI 0—1	INT 0—1
3	DI 0—2	INT 0—2
4	DI 0—3	INT 0—3
5	DI 0—4	INT 0—4
...	...	...
13	DI 0—12	INT 0—12
14	DI 0—13	INT 0—13
15	DI 0—14	INT 0—14

【図３６７】

EITでポップされた EIT.INF中のTYPE	EIT復帰後に受け付けられる EITのTYPE
1	1—4
2	3—4(2—4ではない)
3	3—4
4	4

【図３６３】

No.	Addr	名前	内容	分類	タイプ	スタック
06	+030	予約				
：	：	：				
0F	+078	予約				
10	+080	DBE	デバッグ例外	完了	2	2
11	+088	BAE	ﾊﾞｽｱｸｾｽ例外(ｽﾄｱﾊﾞｯﾌｧ)	完了	1	1
			ﾊﾞｽｱｸｾｽ例外(ｽﾄｱﾊﾞｯﾌｧ以外)	再実行	4	1
12	+090	ATRE	ｱﾄﾞﾚｽ変換例外(ｽﾄｱﾊﾞｯﾌｧ)	完了	1	1
			ｱﾄﾞﾚｽ変換例外(ｽﾄｱﾊﾞｯﾌｧ以外)	再実行	4	1
13	+098	予約	---ﾍﾟｰｼﾞ不在例外			
14	+0A0	PIE	予約命令例外	再実行	4	0
15	+0A8	PIVE	特権命令違反例外	再実行	4	0
16	+0B0	REE	予約領域例外	再実行	4	0
17	+0B8	RSFE	予約ｽﾀｯｸﾌｫｰﾏｯﾄ例外	再実行	4	0
18	+0C0	予約	---ﾘﾝｸﾞ遷移違反例外			
19	+0C8	OAJE	奇数ｱﾄﾞﾚｽｼﾞｬﾝﾌﾟ例外	完了	1	2
1A	+0D0	ZDE	ｾﾞﾛ除算例外	完了	1	2
1B	+0D8	IOE	不正ｵﾍﾟﾗﾝﾄﾞ例外	再実行	4	0
1C	+0E0	予約	---10進不正ｵﾍﾟﾗﾝﾄﾞ例外			
1D	+0E8	LIE	<<L1>>命令例外	再実行	4	0
1E	+0F0	予約				
1F	+0F8	TRAP	条件ﾄﾗｯﾌﾟ命令	完了	1	2
20	+100	TRAP	ﾄﾗｯﾌﾟ命令	完了	1	2
21	+108	TRAP	ﾄﾗｯﾌﾟ命令	完了	1	2
：	：	：				
2F	+178	TRAP	ﾄﾗｯﾌﾟ命令	完了	1	2

【図３６４】

30	+180	CIE	ｺﾌﾟﾛｾｯｻ命令例外	再実行	4	0
31	+188	CIE	ｺﾌﾟﾛｾｯｻ命令例外	再実行	4	0
：	：	：				
37	+1B8	CIE	ｺﾌﾟﾛｾｯｻ命令例外	再実行	4	0
38	+1C0	予約	---ｺﾌﾟﾛｾｯｻ実行例外			
39	+1C8	予約	---ｺﾌﾟﾛｾｯｻｺﾏﾝﾄﾞ例外			
3A	+1D0	予約				
：	：	：				
3F	+1F8	予約				
40	+200	FVEI	固定ﾍﾞｸﾄﾙ外部割り込み	完了	3	0
41	+208	FVEI	固定ﾍﾞｸﾄﾙ外部割り込み	完了	3	0
：	：	：				
46	+230	FVEI	固定ﾍﾞｸﾄﾙ外部割り込み	完了	3	0
47	+238	予約	---固定ﾍﾞｸﾄﾙ外部割り込み			
：	+240	：	固定ﾍﾞｸﾄﾙ外部割り込み			
4E	+277	予約	---固定ﾍﾞｸﾄﾙ外部割り込み			
4F	+278	予約				
50	+280	DI	遅延割り込み例外	完了	3	0
51	+288	DI	遅延割り込み例外	完了	3	0
：	：	：				
5E	+2F0	DI	遅延割り込み例外	完了	3	0
5F	+2F8	予約	---遅延ｺﾝﾃｷｽﾄ例外			
60	+300	予約				
：	：	：				
7F	+3F8	予約				
80	+400	EI	外部割り込み	完了	3	0
：	：	：				
FF	+7F8	EI	外部割り込み	完了	3	0
100	+800	予約				
：	：	：				
1FF	+FF8	予約				

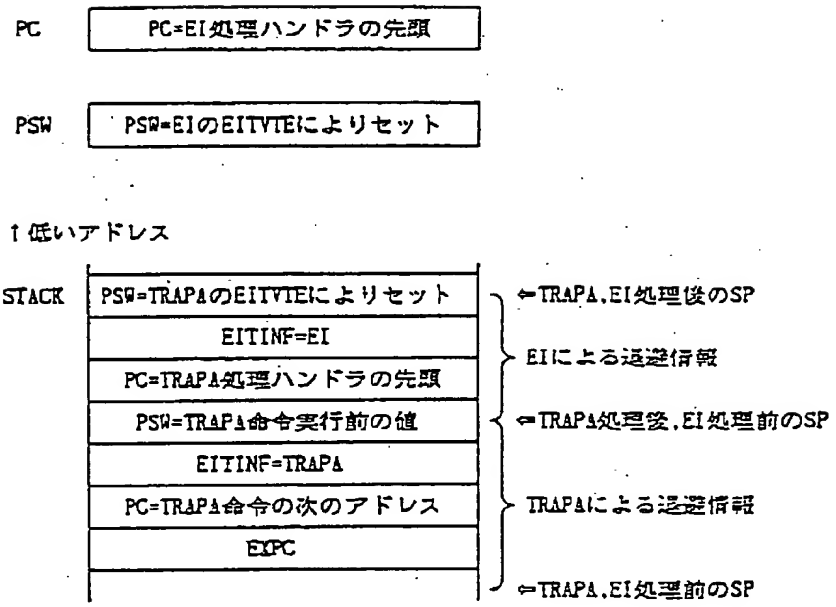
【図３６５】

	〔JRNGでring0に入る場合〕	〔TRAPAでring0に入る場合〕
命令実行 ==)	JRNGの処理	↓
	PC,PSWをSP0/SPIに退避	↓
	↓	↓
	(退避中のエラー発生の場合)	↓
	↓	↓
	命令再実行を正しく行なうため	↓
	JRNGにより退避された	↓
	SP0/SPI上のPC,PSWを除去	↓
	↓	↓
EIT処理 ==)	ページ不在例外などのEIT起動	TRAPAのEIT起動
	PC,PSWをSP0/SPIに退避	PC,PSWをSP0/SPIに退避
	↓	↓
	(退避中のエラー発生の場合)	(退避中のエラー発生の場合)
	↓	↓
EIT中のEIT==)	システムエラー	システムエラー
	動作停止	動作停止

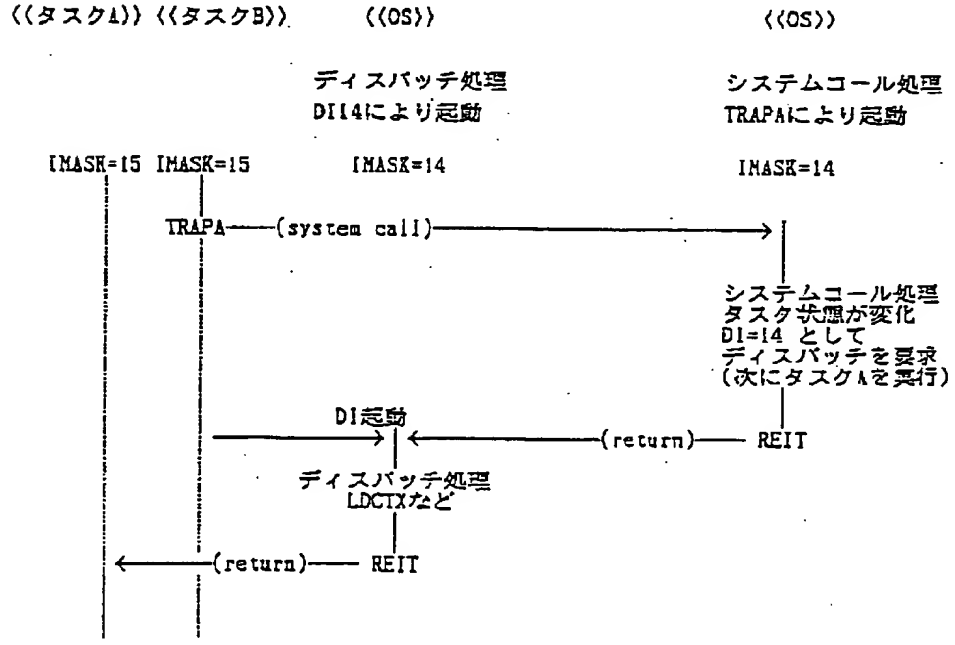
【図４００】

命令		src ==(PACK,UNPK)== dest		実行後SP
PACKWH	@SP+, @-SP	@(initSP)	@(initSP+4-2)	initSP+2
UNPKHW	@SP+, @-SP	@(initSP)	@(initSP+2-4)	initSP-2

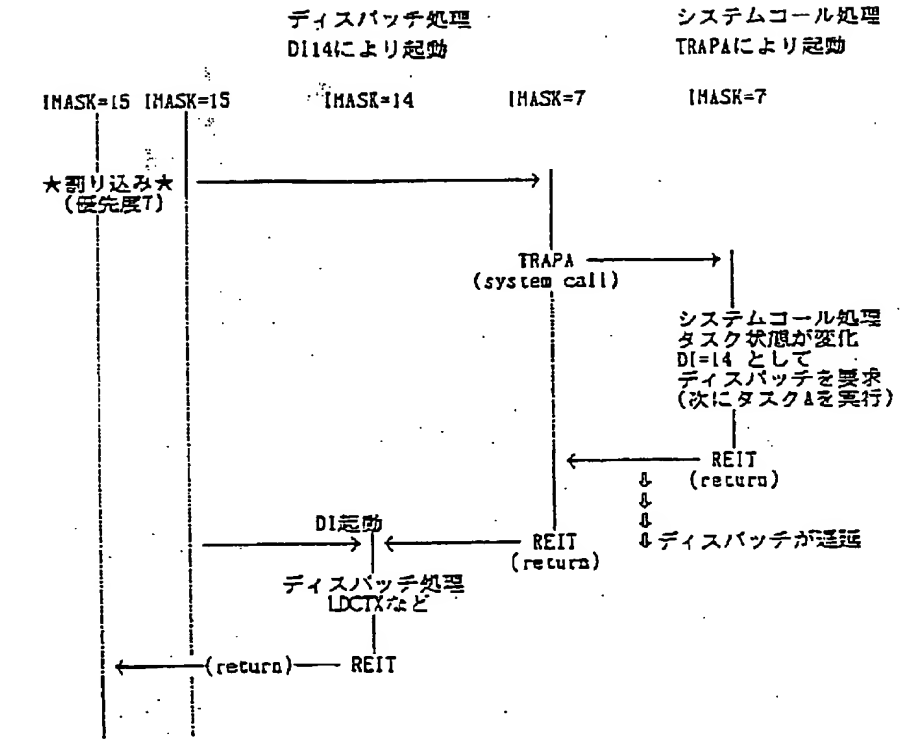
【図３６６】


PC
PC=EI処理ハンドラの先頭
PSW
PSW=EIのEITVTEによりセット
↑低いアドレス
STACK
PSW=TRAPAのEITVTEによりセット
EITINF=EI
PC=TRAPA処理ハンドラの先頭
PSW=TRAPA命令実行前の値
EITINF=TRAPA
PC=TRAPA命令の次のアドレス
EXPC
⇐TRAPA,EI処理後のSP
EIによる退避情報
⇐TRAPA処理後,EI処理前のSP
TRAPAによる退避情報
⇐TRAPA,EI処理前のSP
↓低いアドレス

【図３６９】


〈〈タスクA〉〉
〈〈タスクB〉〉
〈〈OS〉〉
ディスパッチ処理
DI14により起動
〈〈OS〉〉
システムコール処理
TRAPAにより起動
IMASK=15
IMASK=15
IMASK=14
IMASK=14
TRAPA
(system call)
システムコール処理
タスク状態が変化
DI=14 として
ディスパッチを要求
(次にタスクAを実行)
DI起動
(return)
REIT
ディスパッチ処理
LDCTXなど
(return)
REIT

【図３７０】



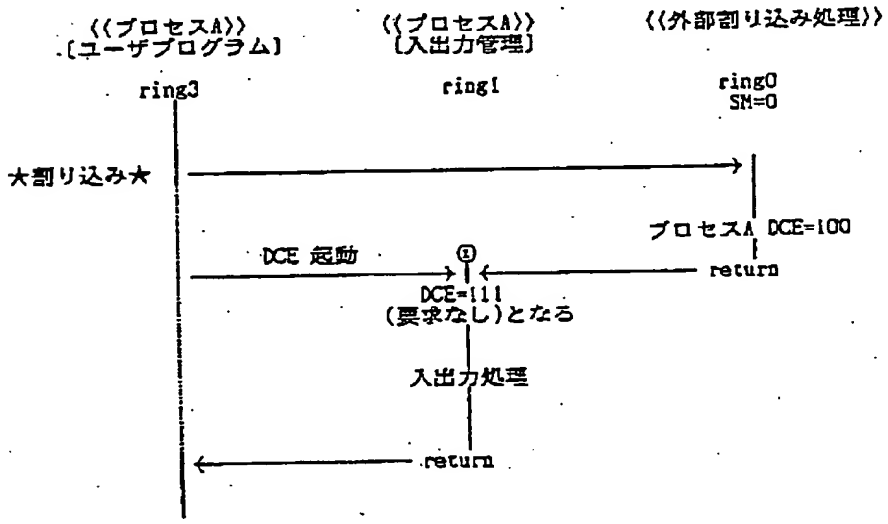
【図３７１】

DCE の値	意味
000	無条件 DCE 要求。SM=1 になれば無条件で DCE が起動される。
001	(reserved)
010	(reserved)
011	(reserved)
100	ring1→ring3 になった時に起動される DCE 要求
101	ring2→ring3 になった時に起動される DCE 要求
110	ring3 になった時に起動される DCE 要求
111	DCE 要求なし

【図３７２】

DCE	DI	外部割り込み(EI)
SMRNGの値により pendingとなる	IMASKの値により pendingとなる	IMASKの値により pendingとなる
コンテキスト従属	コンテキスト独立	コンテキスト独立
内部事象と コンテキストとの関係 (ソフトウエア)	内部事象と プロセッサとの関係 (ソフトウエア)	外部事象と プロセッサとの関係 (ハードウエア)

【図３７３】



【図３８１】

(RIE-X)	11010ORR ..EaR...	11100**+X ..Ea?bf.	**********	**********
(RIE-X)	10111111 ..#ib...	11100**+X ..Ea?bf.	**********	**********
BFEXT:G	11010ORR ..EaR...	111010+X ..EaRbf.	**RRXw**	****RWXd
BFEXT:E	10111111 ..#ib...	111010+X ..EaRbf.	.#6n.. **	****RWXd
BFEXTU:G	11010ORR ..EaR...	111011+X ..EaRbf.	**RRXw**	****RWXd
BFEXTU:E	10111111 ..#ib...	111011+X ..EaRbf.	.#6n.. **	****RWXd
ACB:G	11010ORR ..EaR...	11110PXX ..EaRX..	**RgMXSS	..#dS8..
ACB:E	10111111 ..#ib...	11110PXX ..EaRX..	**RgMXSS	..#dS8..
SCB:G	11010ORR ..EaR...	11111PXX ..EaRX..	**RgMXSS	..#dS8..
SCB:E	10111111 ..#ib...	11111PXX ..EaRX..	**RgMXSS	..#dS8..

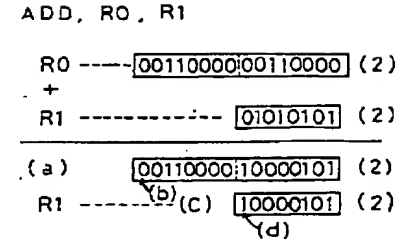
【図３７４】

```
[00??????]

      CMP:L          00RgRwRR 00.ShR..
      MOV:L          00RgMwRR 01.ShR..
      MOV:S          00RgRwWW 10.ShW..
      ****** CMP と MOV が並ぶ場合は、CMP に 0、MOV に 1 が割り当てられる。

      AND:R          00RgMw00 1100RgRw
      OR:R           00RgMw01 1100RgRw
      XOR:R          00RgMw10 1100RgRw
      MOVA:R         00RgWP11 1100RgRP  ......#d16.....
      MUL:R          00RgMw00 1101RgRw
      DIV:R          00RgMw01 1101RgRw

      ⑤拡命令        00????1? 1101????
      ⑥拡命令        00?????? 1111????

[01??????]

      CMP:Q          010#3nRR 00.ShR!I
      MOV:Q          011#3nWW 00.ShW..
      ADD:Q          010#3nMM 01.ShM..
      SUB:Q          011#3nMM 01.ShM..
      SHL:Q          010#3nMM 10.ShM..
      SHL:C          011#3cMM 10.ShM..
      SHA:C          011#3cMM 11.ShM..

      CMP:I          010000RR 11.SbR!I  ......#IR.......
      ADD:I          010001MM 11.ShM..  ......#IM.......
      MOV:I          010010WW 11.ShW..  ......#IW.......
      SUB:I          010011MM 11.ShM..  ......#IM.......
      AND:I          010100MM 11.ShM..  ......#IM.......
      OR:I           010101MM 11.ShM..  ......#IM.......
      XOR:I          010110MM 11.ShM..  ......#IM.......
      [RIE]          010111MM 11.ShM..  ......#IM.......
      ****** CMP と MOV の区別、および ADD と SUB の区別は、2^3 ビットで行なう。また、AND、
            OR、XOR の区別は 2^2～2^3 ビットで行なう。これに、:I フォーマット、:G フォーマ
            ットとも共通である。
```

【図３９１】

終了条件=検索条件 (=処理続行条件の逆)	オプションのニモニック		N_flag=1 の 条件
< R3.or. ≧ R4	OUTU	out of (unsigned)	≧ R4
= R3.or. = R4	OEQ	or. equal	= R4
< R3.or. ≧ R4	OUT	out of (signed)	≧ R4
= 0	Z	zero	= 0 (always)
= R3.or. = 0	ZE	zero. equal	= 0

【図４１７】



【図３７５】

```
[10??????]

        Bcc:D           10cccc00 ..*d8...

        ADD:L           10RgMw01 00.ShRw.
        SUB:L           10RgMw01 01.ShRw.

        BSET:Q          100#3z01 10.ShMfq
        BCLR:Q          101#3z01 10.ShMfq

        BSET1:Q         100#3z01 11ShMfqi
        BIST:Q          101#3z01 11.ShRfq

===== 以下の部分のビット割り当て方針
        10??701?        Ea 有り
        10xx011?        Ea 有り (LDCTX)
        10??111?        imm8 or disp8 有り

        10001?1X        ENTER:E,STM (レジスタリスト有り) ── ENTER:G も共通
        10011?1X        EXIT:E,LDM (レジスタリスト有り) ── EXIT:G も共通

        JRNG:E と JRNG:G、BSR:8 と JSR のパターンもできる限り共通化した。

        JMP             10000010 ..EaA...
        ACS             10000011 ..EaA...
        POP             10010019 ..EaWL..
        PUSHA           10100015 ..EaA...
        PUSH            10110011 ..EaRL..

        LDCTX           10xx0110 ..EaA!A.
        (RIE)           10==0111 ..EaA!A.

        STM             10001019 ..EaWmL.   ......LsWL......
        LDM             10011011 ..EaRmL.   ......L!RL......
        JSR             10101010 ..EaA...
        JRNG:G          10111010 ..EaRh!M
```

【図３７６】

```
        ENTER:E          10001111 ..#ib...  ......LnXL......
        EXITD:E          10011111 ..#ib...  ......LxXL......
        BRA:D            10101110 ..#d8...
        BSR:D            10101111 ..#d8...
        JRNG:E           10111110 ..#ib...
        ①一般形#         10111111 ..#ib...  ???????? ..Ea?...

[11??????]

        CMP:Z            11000OSS ..EaRII.
        MOV:Z            11000lWW ..EaW...
        NEG              11001OMM ..EaM...
        NOT              11001lMM ..EaM...

        ②一般形A         11010ORR ..EaR...  ???????? ..Ea?...
        ③一般形B         11010lRR ..EaR...  ???????? ..Ea?...

        ④一般形特殊      11011000 ..EaA...  ???????? ..Ea?...
        [RIE]            11011001 ********
        [RIE]            11011101 ********

        LDPSB            11011100 ..EaRh..
```

【図３８２】

```
[③一般形B]
        ③一般形B          11010lRR ..EaR...  ???????? ..Ea?...
        ****** 第二HWの割り当て方針
                00<Rn>??      第一HW 'RR'、サイズ指定なし、レジスタ指定あり
                01????SS      第一HW 'RR'、サイズ指定あり、レジスタ指定なし
                10????0?      第一HW 'IR'、サイズ指定なし、レジスタ指定なし
                10<Rn>1?      第一HW 'IR'、サイズ指定なし、レジスタ指定あり
                11<Rn>SS      第一HW 'IR'、サイズ指定あり、レジスタ指定あり(INDEX)

        CSI              11010lRR ..EaR...  00RMC.00 ..EaM1R.
        [RIE-X]          11010lRR ..EaR...  00******01 ..Ea?...
        CHK              11010lRR ..EaR...  00RgWR1c ..EaRdR.

        RVBY             11010lRR ..EaR...  010000WW ..EaW...
        RVBI             11010lRR ..EaR...  010001WW ..EaW...
        PACKss           11010lRR ..EaR...  010010WW ..EaW...
        UNPKss           11010lRR ..EaR...  010011WW ..EaW...  ......#1W.......
        BSCH             11010lRR ..EaR...  0101bdMM ..EaM...
        DCADJ            11010lRR ..EaR...  011000WW ..EaW...
        DCADJU           11010lRR ..EaR...  011001WW ..EaW...
        [RIE-Y]          11010lRR ..EaR...  011010WW ..EaW...
        DCADJX           11010lRR ..EaR...  011011WW ..EaW...
        [RIE-X]          11010lRR ..EaR...  0111****?? ..Ea?...
        ****** DC???X 命令のビットパターンを ??1011SS という形に揃えた
```

【図３７７】

```
        LDPSM           11011101 ..EaRh..
        STPSB           11011110 ..EaWh..
        STPSM           11011111 ..EaWh..

        coproc1         1110****?? ..Ea?...  ********** **********
        coproc2         1111******** **********

〔①② 一般形#/一般形A〕
        ②一般形A        110100RR ..EaR...  ???????? ..Ea?...
        ①一般形#        10111111 ..#ib...  ???????? ..Ea?...

        ADD:G           110100RR ..EaR...  000000MM ..EaM...
        ADD:E           10111111 ..#ib...  000000MM ..EaM...
        ADDU:G          110100RR ..EaR...  000001MM ..EaM...
        ADDU:E          10111111 ..#ib...  000001MM ..EaM...
        SUB:G           110100RR ..EaR...  000010MM ..EaM...
        SUB:E           10111111 ..#ib...  000010MM ..EaM...
        SUBU:G          110100RR ..EaR...  000011MM ..EaM...
        SUBU:E          10111111 ..#ib...  000011MM ..EaM...
        ***** signed の命令と unsigned 命令の区別は、2^2 のビットによって行なわれる。これは、
              ADD, SUB, MUL, DIV, REM, CMP, MOV, BFCMP, BFINS に共通である。

        ADDX:G          110100RR ..EaR...  000100MM ..EaM...
        ADDX:E          10111111 ..#ib...  000100MM ..EaM...
        ADDDX:G         110100RR ..EaR...  000101MM ..EaM...
        ADDDX:E         10111111 ..#ib...  000101MM ..EaM...
        SUBX:G          110100RR ..EaR...  000110MM ..EaM...
        SUBX:E          10111111 ..#ib...  000110MM ..EaM...
        SUBDX:G         110100RR ..EaR...  000111MM ..EaM...
        SUBDX:E         10111111 ..#ib...  000111MM ..EaM...
```

【図３８５】

```
PSTLB           000000← 1111P1W1  0—pttt— ..EaA...
(RIE-X)         000000+X 1111P111  1****0*?? ..Ea?...
SHXL            000000+X 1111-111  1—010— ..EaMX..
SHXR            000000+X 1111-111  1—110— ..EaMX..
ENTER:G         000000+X 1111P111  1—011SS ..EaR!M. ......LnXL......
EXITD:G         000000+X 1111P111  1—111SS ..EaR!M. ......LxXL......
***** EaR!M では Rn と #imm_data のみが許される。EaR!M のサイズは SS で指定される。
      LnRL, LxRL で退避、復帰するレジスタのサイズは X で指定される。

BVPAT           000001+X 1111P111
(RIE)           00001*+X 1111P111
BVSCR           0001bd+X 1111P111

BRA:G           00100DSS 1111P111  ......#dS.......
SSTR            00100ISS 1111P111
BSR:G           00101DSS 1111P111  ......#dS.......
BVMAP           00116Q0X 1111P111
BVCPY           00116Q1X 1111P111
```

【図３７８】

AND:G	110100RR ..EaR...	001000MM ..EaM...
AND:E	10111111 ..#ib...	001000MM ..EaM...
OR:G	110100RR ..EaR...	001001MM ..EaM...
OR:E	10111111 ..#ib...	001001MM ..EaM...
XOR:G	110100RR ..EaR...	001010MM ..EaM...
XOR:E	10111111 ..#ib...	001010MM ..EaM...
DCX:G	110100RR ..EaR...	001011MM ..EaM...
DCX:E	10111111 ..#ib...	001011MM ..EaM...
SHL:G	110100RR ..EaR...	001100MM ..EaM...
SHL:E	10111111 ..#ib...	001100MM ..EaM...
SHA:G	110100RR ..EaR...	001101MM ..EaM...
SHA:E	10111111 ..#ib...	001101MM ..EaM...
ROT:G	110100RR ..EaR...	001110MM ..EaM...
ROT:E	10111111 ..#ib...	001110MM ..EaM...
(RIE-X)	110100RR ..EaR...	001111MM ..EaM...
(RIE-X)	10111111 ..#ib...	001111MM ..EaM...
MUL:G	110100RR ..EaR...	010000MM ..EaM...
MUL:E	10111111 ..#ib...	010000MM ..EaM...
MULU:G	110100RR ..EaR...	010001MM ..EaM...
MULU:E	10111111 ..#ib...	010001MM ..EaM...
DIV:G	110100RR ..EaR...	010010MM ..EaM...
DIV:E	10111111 ..#ib...	010010MM ..EaM...
DIVU:G	110100RR ..EaR...	010011MM ..EaM...
DIVU:E	10111111 ..#ib...	010011MM ..EaM...
(RIE-X)	110100RR ..EaR...	01010*MM ..EaM...
(RIE-X)	10111111 ..#ib...	01010*MM ..EaM...
REM:G	110100RR ..EaR...	010110MM ..EaM...
REM:E	10111111 ..#ib...	010110MM ..EaM...
REMU:G	110100RR ..EaR...	010111MM ..EaM...
REMU:E	10111111 ..#ib...	010111MM ..EaM...

＊＊＊＊＊＊ REM、REMU と DIV、DIVU のパターンをできるだけ共通化した。

【図３９７】

命令		動作	実行後SP
PUSHA	@SP	srcaddr: initSP initSP ==) @(initSP-4)	initSP-4
	(MOVA @SP, @-SP と同じ)		
JSR	@SP	newpc: initSP initPC ==) @(initSP-4) initSP ==) PC	initSP-4
	(initSP-4 ==) PC ではない)		

【図３７９】

```
DCADD:G          11010ORR ..EaR...  011000MM ..EaM...
DCADD:E          10111111 ..#ib...  011000MM ..EaM...
DCADDU:G         11010ORR ..EaR...  011001MM ..EaM...
DCADDU:E         10111111 ..#ib...  011001MM ..EaM...
DCSUB:G          11010ORR ..EaR...  011010MM ..EaM...
DCSUB:E          10111111 ..#ib...  011010MM ..EaM...
DCSUBU:G         11010ORR ..EaR...  011011MM ..EaM...
DCSUBU:E         10111111 ..#ib...  011011MM ..EaM...

(RIE-X)          11010ORR ..EaR...  0111**MM ..EaM...
(RIE-X)          10111111 ..#ib...  0111**MM ..EaM...

CMP:G            11010ORR ..EaR...  100000SS ..EaR!I.
CMP:E            10111111 ..#ib...  100000SS ..EaR!I.
CMPU:G           11010ORR ..EaR...  100001SS ..EaR!I.
CMPU:E           10111111 ..#ib...  100001SS ..EaR!I.
MOV:G            11010ORR ..EaR...  100010WW ..EaW...
MOV:E            10111111 ..#ib...  100010WW ..EaW...
MOVU:G           11010ORR ..EaR...  100011WW ..EaW...
MOVU:E           10111111 ..#ib...  100011WW ..EaW...
```

＊＊＊＊＊＊＊ CMP, CMPU, BFCMP, BFCMPU, DCCMP, DCCMPU および MOV, MOVU, LDP, LDC, BFINS, BFINSU のパターンをできるだけ揃えた。

```
DCCMP:G          11010ORR ..EaR...  100100SS ..EaR!I.
DCCMP:E          10111111 ..#ib...  100100SS ..EaR!I.
DCCMPU:G         11010ORR ..EaR...  100101SS ..EaR!I.
DCCMPU:E         10111111 ..#ib...  100101SS ..EaR!I.
LDC:G            11010ORR ..EaR...  100110WW ..EaW%..
LDC:E            10111111 ..#ib...  100110WW ..EaW%..
LDP:G            11010ORR ..EaR...  100111WW ..EaW%..
LDP:E            10111111 ..#ib...  100111WW ..EaW%..
```

＊＊＊＊＊＊＊ 特殊空間の区別(LDPとLDC)に 2^2 ビットで行なう。STPとSTCの区別も同じである。

【図３９６】

命令		動作	実行後SP
PUSH	SP	src: initSP initSP ==) @(initSP-4)	initSP-4
	(フラグ変化を除けば MOV SP,@-SP と同じ)		
PUSH	@SP+	((モデル上は以下の動作をするが、実際は RIE))	
		src: @initSP @initSP ==) @(initSP+4-4)	initSP
	(フラグ変化を除けば MOV @SP+,@-SP と同じ)		
PUSH	@(d,SP)	src: @(d+initSP) @(d+initSP) ==) @(initSP-4)	initSP-4
	(フラグ変化を除けば MOV @(d,SP),@-SP と同じ)		

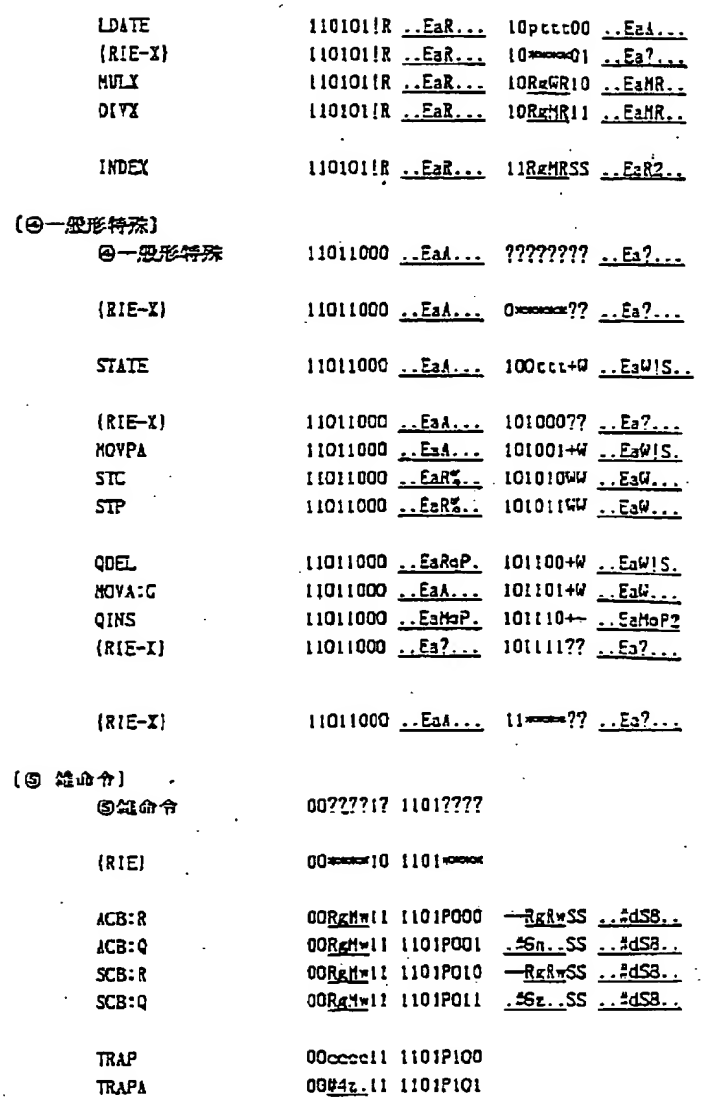
【図３８０】

```
BSETI:G          11010ORR ..EaR...  10100BB ..EaMfi.
BSETI:E          10111111 ..#ib...  10100BB ..EaMfi.
BCLRI:G          11010ORR ..EaR...  10100BB ..EaMfi.
BCLRI:E          10111111 ..#ib...  10100BB ..EaMfi.
(RIE-X)          11010ORR ..EaR...  10101?? ..Ea?...
(RIE-X)          10111111 ..#ib...  10101?? ..Ea?...
DCCMPX:G         11010ORR ..EaR...  10101SS ..EaR!I.
DCCMPX:E         10111111 ..#ib...  10101SS ..EaR!I.

BSET:G           11010ORR ..EaR...  10110BB ..EaMf..
BSET:E           10111111 ..#ib...  10110BB ..EaMf..
BCLR:G           11010ORR ..EaR...  10110BB ..EaMf..
BCLR:E           10111111 ..#ib...  10110BB ..EaMf..
BNOT:G           11010ORR ..EaR...  10111BB ..EaMf..
BNOT:E           10111111 ..#ib...  10111BB ..EaMf..
BTST:G           11010ORR ..EaR...  10111BB ..EaRf..
BTST:E           10111111 ..#ib...  10111BB ..EaRf..

BFCMP:G:R        110100RR ..EaR...  11000+X ..EaRbf.  **RRXw**  *****RRXs
BFCMP:E:R        10111111 ..#ib...  11000+X ..EaRbf.  .#6n..**  *****RRXs
BFCMPU:G:R       110100RR ..EaR...  11001+X ..EaRbf.  **RRXw**  *****RRXs
BFCMPU:E:R       10111111 ..#ib...  11001+X ..EaRbf.  .#6n..**  *****RRXs
BFINS:G:R        110100RR ..EaR...  11010+X ..EaMbf.  **RRXw**  *****RRXs
BFINS:E:R        10111111 ..#ib...  11010+X ..EaMbf.  .#6n..**  *****RRXs
BFINSU:G:R       110100RR ..EaR...  11011+X ..EaMbf.  **RRXw**  *****RRXs
BFINSU:E:R       10111111 ..#ib...  11011+X ..EaMbf.  .#6n..**  *****RRXs

BFCMP:G:I        110100RR ..EaR...  11010O+X ..EaRbf.  **RRXwSS ..#iS8..
BFCMP:E:I        10111111 ..#ib...  11010O+X ..EaRbf.  .#6n..SS ..#iS8..
BFCMPU:G:I       110100RR ..EaR...  11010I+X ..EaRbf.  **RRXwSS ..#iS8..
BFCMPU:E:I       10111111 ..#ib...  11010I+X ..EaRbf.  .#6n..SS ..#iS8..
BFINS:G:I        110100RR ..EaR...  11011O+X ..EaMbf.  **RRXwSS ..#iS8..
BFINS:E:I        10111111 ..#ib...  11011O+X ..EaMbf.  .#6n..SS ..#iS8..
BFINSU:G:I       110100RR ..EaR...  11011I+X ..EaMbf.  **RRXwSS ..#iS8..
BFINSU:E:I       10111111 ..#ib...  11011I+X ..EaMbf.  .#6n..SS ..#iS8..
```

【図３９９】

命令		src		dest	実行後SP
MOV.W	@SP+, @-SP	@initSP	==>	@initSP	initSP
MOV.W	@SP+, @(d,SP)	@initSP	==>	@(d+initSP+4)	initSP+4
MOV.W	@(d,SP), @-SP	@(d+initSP)	==>	@(initSP-4)	initSP-4
MOV.W	SP, @-SP	initSP	==>	@(initSP-4)	initSP-4
MOV.W	@SP+, SP	@initSP	==>	SP	@initSP

【図３８３】



【図４０３】

命令		src1	src2	実行後SP
CMP.W	@SP+, @SP+	@(initSP)	@(initSP+4)	initSP+8
CMP.W	@SP+, @(d,SP)	@(initSP)	@(d+initSP+4)	initSP+4
CMP.W	@SP+, SP	@(initSP)	initSP+4	initSP+4
CMP.W	SP, @SP+	initSP	@(initSP)	initSP+4

【図３８４】

```
xxxxx 以下の部分の割り当て方針
        00??0011 1101P110      LVreserved
        00??1011 1101P110      一般命令
        00??0111 1101P110      特権命令（STCTX）
        00??1111 1101P110      特権命令

LVreserved          00xx0011 1101*110
STCTX               00xx0111 1101P110

PIB                 00001011 1101P110
NOP                 00011011 1101-110
RTS                 00101011 1101P110
RRNG                00111011 1101P110

WAIT                00001111 1101-110   ......#ih.......
REIT                00101111 1101P110
PTLB                00p11111 1101P110

(RIE)               00xxxx11 1101P111

(© 雑命令)
        ©雑命令     00?????? 11??????

        SCMP        00eeeeSS 1110P00b
        SMOV        00eeeeSS 1110P10b
        SCH         00eeeeSS 1111P0ab
        SSCH        00eeeeSS 1111P10c

        Bcc:G       00ccccSS 1111P110   ......$dS.......

xxxxx 以下の部分の割り当て方針
        000????? 1111P111       第一HW の 2^9 のビットは常に '+'
        001????? 1111P111       第一HW の 2^9 のビットは 0/1 の両方が可
```

【図４０１】

命令		動作	実行後SP
MOV.W	SP, @-SP	initSP ==) @(initSP-4)	initSP-4
STC.W	@spO, @-SP		
	((モデル上は以下の動作をするが、実際はインプリメント依存))		
		initSP-4 ==) @(initSP-4)	initSP-4
	(spO は SPO を制御レジスタとして指定したもの)		
	(initSP ==) @(initSP-4) ではない)		
MOVA.W	@SP, @-SP	initSP ==) @(initSP-4)	initSP-4
	(initSP-4 ==) @(initSP-4) ではない)		

【図３８６】

```
#3c     SHA:C, SHL:C
#3n     ADD:Q, CMP:Q, MOV:Q, SHL:Q, SUB:Q
#3z     BCLR:Q, BSET:Q, BSETI:Q, BTST:Q
#4z     IRAPA
#6n     ACB:Q, BFCMP:E:I, BFCMP:E:R, BFCMPU:E:I, BFCMPU:E:R, BFEXT:E, BFEXTU:E, BFINS:E:I,
        BFINS:E:R, BFINSU:E:I, BFINSU:E:R
#6z     SCB:Q
#d16    MOVA:B
#d8     BRA:D, BSR:D, Bcc:D
#dS     BRA:G, BSR:G, Bcc:G
#dS8    ACB:E, ACB:G, ACB:Q, ACB:R, SCB:E, SCB:G, SCB:Q, SCB:R
#iM     ADD:I, AND:I, OR:I, SUB:I, XOR:I, (RIE)
#iR     CMP:I
#iS8    BFCMP:E:I, BFCMP:G:I, BFCMPU:E:I, BFCMPU:G:I, BFINS:E:I, BFINS:G:I, BFINSU:E:I,
        BFINSU:G:I
#iW     MOV:I, UNPKss
#ib     ACB:E, ADD:E, ADDDX:E, ADDU:E, ADDX:E, AND:E, BCLR:E, BCLRI:E, BFCMP:E:I, BFCMP:E:R,
        BFCMPU:E:I, BFCMPU:E:R, BFEXT:E, BFEXTU:E, BFINS:E:I, BFINS:E:R, BFINSU:E:I, BFINSU:E:
        R, BNOT:E, BSET:E, BSETI:E, BTST:E, CMP:E, CMPU:E, DCADD:E, DCADDU:E, DCCMP:E, DCCMPU:
        E, DCCMPX:E, DCSUB:E, DCSUBU:E, DCX:E, DIV:E, DIVU:E, ENTER:E, EXITD:E, JRNG:E, LDC:E,
        LDP:E, MOV:E, MOVU:E, MUL:E, MULU:E, OR:E, REM:E, REMU:E, ROT:E, SCB:E, SHA:E, SHL:E,
        SUB:E, SUBDX:E, SUBU:E, SUBX:E, XOR:E
#ih     WAIT
EaA     ACS, JMP, JSR, LDATE, MOVA:C, MOVPA, PSTLB, PUSHA, STATE
EaA!A   LDCTX, (RIE)
EaM     ADD:E, ADD:G, ADDDX:E, ADDDX:G, ADDU:E, ADDU:G, ADDX:E, ADDX:G, AND:E, AND:G, BSCH,
        DCADD:E, DCADD:G, DCADDU:E, DCADDU:G, DCSUB:E, DCSUB:G, DCSUBU:E, DCSUBU:G, DCX:E,
        DCX:G, DIV:E, DIV:G, DIVU:E, DIVU:G, MUL:E, MUL:G, MULU:E, MULU:G, NEG, NOT, OR:E, O
        G, REM:E, REM:G, REMU:E, REMU:G, ROT:E, ROT:G, SHA:E, SHA:G, SHL:E, SHL:G, SUB:E, SUB:
        G, SUBDX:E, SUBDX:G, SUBU:E, SUBU:G, SUBX:E, SUBX:G, XOR:E, XOR:G
```

【図３８９】

ニモニック	意味	条件	cccc
XS	X_flag set	X	0000
XC	X_flag clear	¬X	0001
EQ	equal / Z_flag set	Z	0010
NE	not equal / Z_flag clear	¬Z	0011
LT	less than / L_flag set	L	0100
GE	greater or equal / L_flag clear	¬L	0101
LE	less or equal	L+Z	0110
GT	greater than	¬L*¬Z	0111
VS	V_flag set	V	1000
VC	V_flag clear	¬V	1001
MS	minus / M_flag set	M	1010
MC	plus / M_flag clear	¬M	1011
FS	F_flag set	F	1100
FC	F_flag clear	¬F	1101
(RIE)			1110
(RIE)			1111

【図３８７】

```
EaMR    DIVX, MULX
EaMX    SEXL, SEXR
EaMbf   BFINS:E:I, BFINS:E:R, BFINS:G:I, BFINS:G:R, BFINSU:E:I, BFINSU:E:R, BFINSU:G:I,
        BFINSU:G:R
EaMf    BCLR:E, BCLR:G, BNOT:E, BNOT:G, BSET:E, BSET:G
EaMfi   BCLRI:E, BCLRI:G, BSETI:E, BSETI:G
EaMiR   CSI
EaMqP   QINS
EaMqP2  QINS
EaR     ACB:G, ADD:G, ADDDX:G, ADDU:G, ADDX:G, AND:G, BCLR:G, BCLRI:G, BFCMP:G:I, BFCMP:G:R,
        BFCMPU:G:I, BFCMPU:G:R, BFEXT:G, BFEXTU:G, BFINS:G:I, BFINS:G:R, BFINSU:G:I, BFINSU:G:
        R, BNOT:G, BSCN, BSET:G, BSETI:G, BTST:G, CHK, CMP:G, CMPU:G, CSI, DCADD:G, DCADDU:G,
        DCADJ, DCADJU, DCADJX, DCCMP:G, DCCMPU:G, DCCMPX:G, DCSUB:G, DCSUBU:G, DCX:G, DIV:G,
        DIVU:G, DIVX, INDEX, LDATE, LDC:G, LDP:G, MOV:G, MOVU:G, MUL:G, MULU:G, MULX, OR:G,
        PACKss, REM:G, REMU:G, ROT:G, RVBI, RVBY, SCB:G, SHA:G, SHL:G, SUB:G, SUBDX:G, SUBU:G,
        SUBX:G, UNPKss, XOR:G

EaR!I   CMP:E, CMP:G, CMP:Z, CMPU:E, CMPU:G, DCCMP:E, DCCMP:G, DCCMPU:E, DCCMPU:G, DCCMPX:E,
        DCCMPX:G
EaR!M   ENTER:G, EXITD:G
EaR%    STC, STP
EaR2    INDEX
EaRL    PUSH
EaRX    ACB:E, ACB:G, SCB:E, SCB:G
EaRbf   BFCMP:E:I, BFCMP:E:R, BFCMP:G:I, BFCMP:G:R, BFCMPU:E:I, BFCMPU:E:R, BFCMPU:G:I,
        BFCMPU:G:R, BFEXT:E, BFEXT:G, BFEXTU:E, BFEXTU:G
EaRdR   CHK
EaRf    BTST:E, BTST:G
EaRh    LDPSB, LDPSM
EaRb!M  JRNG:G
EaRmL   LDM
EaRqP   QDEL
EaW     DCADJ, DCADJU, DCADJX, MOV:E, MOV:G, MOV:Z, MOVA:G, MOVU:E, MOVU:G, PACKss, RVBI,
        RVBY, STC, STP, UNPKss
```

【図４０４】

命令	動作	
INDEX	@SP+,@SP+,SP	
	indexsize:	@initSP
	subscript:	@(initSP+4)
	EX.の直前でのxreg:	initSP+8
	(InitSP+8)×@initSP+@(initSP+4) ==> SP	

【図３８８】

```
EaW|S   MOVPA, QDEL, STATE
EaW%    LDC:E, LDC:G, LDP:E, LDP:G
EaWL    POP
EaWb    STPSB, STPSM
EaWmL   STM
LlRL    LDM
LnXL    ENTER:E, ENTER:G
LsWL    STM .
LxXL    EXITD:E, EXITD:G
RMC     CSI
RRXs    BFCMP:E:R, BFCMP:G:R, BFCMPU:E:R, BFCMPU:G:R, BFINS:E:R, BFINS:G:R, BFINSU:E:R,
        BFINSU:G:R
RRXw    BFCMP:G:I, BFCMP:G:R, BFCMPU:G:I, BFCMPU:G:R, BFEXT:G, BFEXTU:G, BFINS:G:I, BFINS:G:R,
        BFINSU:G:I, BFINSU:G:R
RWXd    BFEXT:E, BFEXT:G, BFEXTU:E, BFEXTU:G
RgMR    DIVX, INDEX
RgMX    ACB:E, ACB:G, SCB:E, SCB:G
RgMw    ACB:Q, ACB:R, ADD:L, AND:R, DIV:R, MUL:R, OR:R, SCB:Q, SCB:R, SUB:L, XOR:R
RgRP    MOVA:R
RgRw    ACB:R, AND:R, CMP:L, DIV:R, MOV:S, MUL:R, OR:R, SCB:R, XOR:R
RgWP    MOVA:R
RgWR    CHK, MULX
RgWw    MOV:L
ShM     ADD:I, ADD:Q, AND:I, OR:I, SHA:C, SHL:C, SHL:Q, SUB:I, SUB:Q, XOR:I, (RIE)
ShMfq   BCLR:Q, BSET:Q
ShMfqi  BSETI:Q
ShR     CMP:L, MOV:L
ShR!I   CMP:I, CMP:Q
ShRfq   BTST:Q
ShRw    ADD:L, SUB:L
ShW     MOV:I, MOV:Q, MOV:S
```

【図４０６】

命令		src	destから読む値	実行後SP
ADD.W	@SP+,@(d,SP)	@initSP	@(d+initSP+4)	initSP+4
	(@(d+initSP+4)+@initSP ==) @(d+initSP+4) を実行)			
ADD.W	@SP+,SP	@initSP	initSP+4	(initSP+4) +@initSP
	((initSP+4)+@initSP ==) SP を実行)			

【図３９０】

終了条件=検索条件 (=処理続行条件の逆)	オプションのニモニック			eeee
＜ R3	LTU	less than (unsigned)		0000
≧ R3	GEU	greater or equal (unsigned)		0001
＝ R3	EQ	equal		0010
≠ R3	NE	not equal		0011
＜ R3	LT	less than (signed)		0100
≧ R3	GE	greater or equal (signed)		0101
終了条件なし	N	never (またはオプションなし)		0110
	(RIE)			0111
＜ R3.or. ≧ R4	OUTU	out of (unsigned)	⟨(L2)⟩	1000
≧ R3.and.＜ R4	WINU	within (unsigned)	⟨(L2)⟩	1001
＝ R3.or. ＝ R4	OEQ	or, equal	⟨(L2)⟩	1010
≠ R3.and.≠ R4	ANE	and, not equal	⟨(L2)⟩	1011
＜ R3.or. ≧ R4	OUT	out of (signed)	⟨(L2)⟩	1100
≧ R3.and.＜ R4	WIN	within (signed)	⟨(L2)⟩	1101
＝ 0	Z	zero	⟨(L2)⟩	1110
＝ R3.or. ＝ 0	ZE	zero, equal	⟨(L2)⟩	1111

【図３９２】

src=0, dest=0 の演算結果を bit0 に、
src=0, dest=1 の演算結果を bit1 に、
src=1, dest=0 の演算結果を bit2 に、
src=1, dest=1 の演算結果を bit3 に入れる。

0000	F	False	0 ==⟩ dest
0001	NAN	NotAndNot	~dest .and. ~src ==⟩ dest
0010	AN	AndNot	dest .and. ~src ==⟩ dest
0011	NS	NotSrc	~src ==⟩ dest
0100	NA	NotAnd	~dest .and. src ==⟩ dest
0101	ND	NotDest	~dest ==⟩ dest
0110	X	Xor	dest .xor. src ==⟩ dest
0111	NON	NotOrNot	~dest .or. ~src ==⟩ dest
1000	A	And	dest .and. src ==⟩ dest
1001	NX	NotXor	~dest .xor. src ==⟩ dest
1010	D	Dest	dest ==⟩ dest
1011	ON	OrNot	dest .or. ~src ==⟩ dest
1100	S	Src	src ==⟩ dest
1101	NO	NotOr	~dest .or. src ==⟩ dest
1110	O	Or	dest .or. src ==⟩ dest
1111	T	True	1 ==⟩ dest

【図３９３】

	一般①	Rn	#imm	@SP+	@-SP	付加	(対象命令)
EaA	○	×	×	×	×	○	ACS, JMP, JSR, LDATE, MOVA:G, PUSHA, MOVPA, PSTLB, STATE,
EaA|A	○	×	×	×	×	×	LDCTX
EaM	○	○	×	×	×	○	ADD:E, ADD:G, DIV:E, DIV:G, DIVU:E, DIVU:G, SHA:E, SHA:G.....多数
ShM							ADD:I, ADD:Q, SHA:C, OR:I, AND:I, SHL:Q, SHL:G, SUB:I, SUB:Q, XOR:I
EaMX							SHXL, SHXR
EaMR							DIVX, MULX
EaMf	○	○②	×	×	×	○	BCLR:E, BCLR:G, BNOT:E, BNOT:G, BSET:E, BSET:G
ShMfq	○	○	×	×	×	○	BCLR:Q, BSET:Q
EaMbf	○	⟨(L2)⟩	×	×	×	○	BFINS:E:I, BFINS:E:R, BFINSU:E:I, BFINSU:E:R BFINS:G:I, BFINS:G:R, BFINSU:G:I, BFINSU:G:R
EaMfi	○	×	×	×	×	○	BCLRI:E, BCLRI:G, BSETI:E, BSETI:G
ShMfqi	○	×	×	×	×	○	BSETI:Q
EaMiR	○	×	×	×	×	○	CSI
EaMqP	○	×	×	×	×	○	QINS
EaMqP2							QINS
	一般	Rn	#imm	@SP+	@-SP	付加	(対象命令)

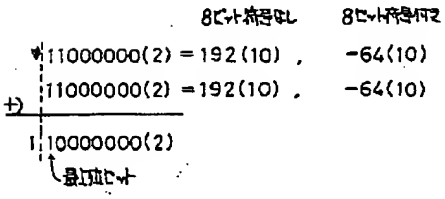
【図４０７】

命令	動作	
[CSI	comp,update,dest]	
CSI	SP,@SP+,@(d,SP)	
	EX.の直前でのcomp:	initSP+4
	update:	@initSP
	dest:	@(d+initSP+4)
	(dest:	@(d+initSP) ではない]

【図３９４】

	一般	Rn	#imm	@SP+	@-SP	付加	(対象命令)
EaR	○	○	○	○	×	○	ACB:G, ADD:G, ADDDX:G, ADDU:G, ADDX:G, AND:G, BCLR:G, BSET:G,...多数
EaRh							LDPSB, LDPSH
ShR							CMP:L, MOV:L
ShRw							ADD:L, SUB:L
EaR2	○	○	○③	○	×	○	INDEX
EaRX							ACB:E, ACB:G, SCB:E, SCB:G
EaRmL	○	×	×	○	×	×	LDM
EaRL	○	○	○	×	×	○	PUSH
EaRII	○	○	×	○	×	○	CMP:E, CMP:G, CMPU:E, CMPU:G, CMP:Z
ShRII							CMP:I, CMP:Q
EaRm	○	×	×	×	×	○	STC, STP
EaRdR	○	×	×	×	×	○	CHK
EaRqP	○	×	×	×	×	○	QDEL
EaRf	○	○②	×	×	×	○	BTST:E, BTST:G
ShRfq	○	○	×	×	×	○	BTST:Q
EaRbf	○	((L2))	×	×	×	○	BFCMP:E:I, BFCMPU:E:I, BFCMP:E:R, BFCMPU:E:R, BFEXT:E, BFEXTU:E, BFCMP:G:I, BFCMPU:G:I, BFCMP:G:R, BFCMPU:G:R, BFEXT:G, BFEXTU:G
EaRIM	×	○	○	×	×	×	ENTER:G, EXITD:G
EaRhIM							JRNG:G
	一般	Rn	#imm	@SP+	@-SP	付加	(対象命令)

【図４１５】



【図３９５】

	一般	Rn	#imm	@SP+	@-SP	付加	(対象命令)
EaW	○	○	×	×	○	○	MOV:Z, MOV:E, MOV:C, MOVA:C, MOVU:E, MOVU:C PACKss, STC, STP, UNPKss, RVBY, RVBI
EaWh							STPSB, STPSH
ShW							MOV:I, MOV:Q, MOV:S
EaW!S	○	○	×	×	×	○	MOVPA, STATE, QDEL
EaWmL	○	×	×	×	○	×	STM
EaW%	○	×	×	×	×	○	LDC:E, LDC:C, LDP:E, LDP:C
EaWL	○	○	×	×	×	○	POP
	一般	Rn	#imm	@SP+	@-SP	付加	(対象命令)

〔注〕
① 「一般」には、@abs, @(disp.PC), @(disp, Rn), @Rn が含まれる。
② レジスタ対象のビット操作命令では、offset は下位ビットのみが有効
③ 第二オペランドのイミディエートなので変則的だが、必要性があるのでインプリメントする。

【図４０５】

命令	動作	
〔ACB	step,xreg,limit,newpc〕	
ACB	@SP+,SP,@SP+,newpc	
	step:	@initSP
	limit:	@(initSP+4)
	命令実行前のxreg(=SP)	initSP
	EX.の直前でのxreg:	initSP+8　(@SP+,@SP+ による)
	命令実行後のxreg:	initSP+8+@initSP
	if((initSP+8+@initSP) < @(initSP+4)) jump to newpc endif	
〔SCB	step,xreg,limit,newpc〕	
SCB	@SP+,SP,@SP+,newpc	
	step:	@initSP
	limit:	@(initSP+4)
	命令実行前のxreg(=SP)	initSP
	EX.の直前でのxreg:	initSP+8　(@SP+,@SP+ による)
	命令実行後のxreg:	initSP+8-@initSP
	if((initSP+8+@initSP) ≧ @(initSP+4)) jump to newpc endif	

【図３９８】

命令	動作	実行後SP
POP	SP　　@initSP ==〉 SP	@initSP
	〔EX で一旦 initSP+4 ==〉 SP となるが、	
	WW1 で @initSP が SP に overwrite される〕	
	〔フラッグ変化を除けば MOV @SP+,SP と同じ〕	
POP	@-SP　〈〈モデル上は以下の動作をするが、実際は RIE〉〉	
	@(initSP-4) ==〉 @(initSP-4)　　initSP	
	〔WA1 で initSP-4 ==〉 SP, woladdr、	
	EX で @(initSP-4) ==〉 dest2、	
	initSP-4+4 ==〉 SP、	
	WW1 で dest2 ==〉 @woladdr、	
	つまり @(initSP-4) ==〉 @(initSP-4)〕	
	〔MOV @SP+,@-SP は @initSP ==〉 @initSP で動作が異なる〕	
POP	@(d,SP)　　@initSP ==〉 @(d+initSP)　　initSP+4	
	〔WA1 で d+initSP ==〉 woladdr、	
	EX で @initSP ==〉 dest2、	
	initSP+4 ==〉 SP、	
	WW1 で dest2 ==〉 @woladdr、	
	つまり @initSP ==〉 @(d+initSP)〕	
	〔MOV @SP+,@(d,SP) は @initSP ==〉 @(d+4+initSP) で動作が異なる〕	

【図４０９】

加算命令(ADD)

		32000	80386	68000	IBM/370	VAX	本発明
符号付き	大小		SF(最上位ビット)	N(最上位ビット)	○	N(最上位ビット)	L(サインビット) M(最上位ビット)
	等		ZF	Z	○	Z	Z
	オーバーフロー	F	OF	V	○	V	V
	桁上げ	C	CF	C		C	X
	その他			X(Cと等しい)			
符号なし	大小						(L=0) M(最上位ビット)
	等		ZF	Z	○	Z	Z
	オーバーフロー						V
	桁上げ						X
	その他						

【図４０２】

命令	動作	実行後SP
[BFINS	src,offset,width,base]	
BFINS	SP,@SP+,SP,@(d,SP)	initSP+4
	src: initSP+4	
	offset: @initSP	
	width: initSP+4	
	base: @(d+initSP+4)	
	initSP+4 ==) bitfield(offset,width,base)	
[BFEXT	offset,width,base,dest]	
BFEXT	@SP+,SP,@(d,SP),R0	initSP+4
	offset: @initSP	
	width: initSP+4	
	base: @(d+initSP+4)	
	bitfield(offset,width,base) ==) R0	
BFEXT	@SP+,SP,@(d,SP),SP	bitfield
	offset: @initSP	
	width: initSP+4	
	base: @(d+initSP+4)	
	bitfield(offset,width,base) ==) SP	
	[EX に入る直前の SP値は initSP+4 であるが、	
	EX で抽出されたビットフィールドが	
	SP に overwrite される]	

【図４１０】

乗算命令(MUL)

		32000	80386	68000	IBM/370	VAX	本発明
符号付き	大小			N(サインビット)		N	L(サインビット) M(最上位ビット)
	等			Z		Z	Z
	オーバフロー		OF	(V=0)		V	V
	桁上げ		CF	(C=0)		(C=0)	
	その他	オーバフローした場合上位ビットが1		(オーバフローしない)	(オーバフローしない)		
符号なし	大小			N(最上位ビット)			(L=0) M(最上位ビット)
	等			Z		Z	Z
	オーバフロー			(V=0)			V
	桁上げ			(C=0)			
	その他			(オーバフローしない)			

【図４０８】

命令		命令実行後のリソース値	
STM.W	(SP), @-SP	initSP-4 ==> SP	(@-SP による)
		initSP ==> @(initSP-4)	
		[initSP-4 ==) @(initSP-4) ではない]	
STM.W	(R0,SP), @-SP	initSP-8 ==> SP	(@-SP による)
		initSP ==) @(initSP-4)	
		[initSP-8 ==) @(initSP-4) ではない]	
		R0 ==) @(initSP-8)	
STM.W	(R0,R1), @-SP	initSP-8 ==> SP	(@-SP による)
		R1 ==) @(InitSP-4)	
		R0 ==) @(initSP-8)	
STM.W	(R0,SP), @SP	initSP ==) SP	
		R0 ==) @(InitSP)	
		initSP ==) @(initSP+4)	
LDM.W	@SP+, (SP)	initSP+4 ==) SP	(ポインタの更新による)
		[@(initSP) の値は捨てられる]	
LDM.W	@SP+, (R0,SP)	initSP+8 ==) SP	(ポインタの更新による)
		@(initSP) ==) R0	
		(@(initSP+4) の値は捨てられる)	
LDM.W	@SP+, (R0,R1)	initSP+8 ==) SP	(ポインタの更新による)
		@(initSP) ==) R0	
		@(initSP+4) ==) R1	
LDM.W	@SP, (R0,SP)	@(initSP+4) ==) SP	(SP の転送による)
		@(initSP) ==) R0	

【図４１１】

減算命令(SUB)

		32000	80386	68000	IBM 370	VAX	本発明
符号付き	大小		SF(最上位ビット)	N(最上位ビット)	○	N(最上位ビット)	L(サインビット) M(最上位ビット)
	等		ZF	Z	○	Z	Z
	オーバフロー	F	OF	V	○	V	V
	桁上げ	C	CF	C		C	X
	その他			X(Cと等しい)			
符号なし	大小						L(サインビット) M(最上位ビット)
	等		ZF	Z	○	Z	Z
	オーバフロー						V
	桁上げ						X
	その他						

【図４１２】

除算命令(DIV)

		32000	80386	68000	IBM 370	VAX	本発明
符号付き	大小			N(最上位ビット)		N(最上位ビット)	L(サインビット) M(最上位ビット)
	等			Z		Z	Z
	オーバーフロー			V(0除算も)		V(0除算も)	V(0除算も)
	桁上げ			(C=0)		(C=0)	
	その他						
符号なし	大小			N(最上位ビット)			(L=0) M(最上位ビット)
	等			Z		Z	Z
	オーバーフロー			V(0除算も)			V(0除算も)
	桁上げ			(C=0)			
	その他						

【図４１３】

データ移動命令(MOV)

		32000	80386	68000	IBM 370	VAX	本発明
符号付き	大小			N(サインビット)		N(サインビット)	M(最上位ビット)
	等			Z		Z	Z
	オーバーフロー			(V=0)		(V=0)	V
	桁上げ			(C=0)			
	その他			(オーバーフローしない)		(オーバーフローしない)	
符号なし	大小						M(最上位ビット)
	等			Z		Z	Z
	オーバーフロー			(V=0)		(V=0)	V
	桁上げ			(C=0)			
	その他			(オーバーフローしない)		(オーバーフローしない)	

【図４１４】

比較命令(CMP)

		32000	80386	68000	IBM/370	VAX	本発明
符号付き	大小	N	SF(最上位ビット)	N(最上位ビット)	○	N(マイナスビット)	L(マイナスビット)
	等	Z	ZF	Z	○	Z	Z
	オーバフロー		OF	V		(V=Ø)	
	桁上げ		CF	C			
	その他					(オーバフローしない)	
符号なし	大小	L				C	L
	等	Z	ZF	Z	○	Z	Z
	オーバフロー					(V=Ø)	
	桁上げ						
	その他					(オーバフローしない)	